

Aztec C Reference Manual for the Amiga

**Version 5.0
October 1989**

Copyright © 1988, 1989 by Manx Software Systems, Inc.
All Rights Reserved
Worldwide

DISTRIBUTED BY:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702
(201) 542-2121

USE RESTRICTIONS

You are permitted to install and use this product on a single computer. Multiple CPU systems require supplementary licenses.

Before using any Aztec C products, the License Registration included with this product must be signed and mailed to:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702

COPYRIGHT

This software package and document are copyrighted ©1988, 1989 by Manx Software Systems. All rights reserved worldwide.

No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language without prior written permission of Manx Software Systems.

DISCLAIMER

Manx Software Systems makes no representations or warranties with respect to this product and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Manx Software Systems reserves the right to modify the programs and revise the contents of the manual without obligation to notify any person of such revision or changes.

TRADEMARKS

Aztec C, Manx AS, Manx LN, Z, and SDB are trademarks of Manx Software Systems. CP/M-86 and CP/M-80 are trademarks of Digital Research. MSDOS is a trademark of Microsoft. PC DOS is a trademark of IBM. UNIX is a trademark of AT&T Bell Laboratories. Macintosh and Apple II are trademarks of Apple Computer. Atari is a trademark of Atari Computers. Amiga is a trademark of Commodore-Amiga.

Manual Revision History

October 1989	Fifth Edition
December 1988	Fourth Edition
May 1988	Third Edition
April 1986	Second Edition
February 1986	First Edition

Table of Contents

Chapter 1 - Overview

Introduction	1-1
ANSI C	1-1

Chapter 2 - Compiler

Using the Compiler	2-1
Input File	2-2
Source Filename Extensions	2-2
The Output Files	2-3
Creating an Object File	2-3
Creating an Assembly Language File	2-4
Searching for #include Files	2-5
Option -i	2-5
#include Search Order	2-6
Pre-compiled #include Files	2-6
Memory Models	2-8
Large Data versus Small Data	2-8
Large Code Versus Small Code	2-9
Selecting a Module's Memory Model	2-12
Libraries	2-12
Multi-module Programs	2-13
Code Segmentation	2-14
Compiler Options	2-14
Option Philosophy	2-14
CCOPTS Environment Variable	2-15
Option List	2-17
Option Descriptions	2-21

Option -3	2-21
Option -5	2-22
Option -c2	2-22
Option -d macro	2-22
Option -fa	2-23
Option -ff	2-23
Option -fm	2-24
Option -f8	2-24
Option -hi filename	2-24
Option -ho filename	2-24
Option -i path	2-25
Option -k	2-25
Option -mb	2-26
Option -mc	2-26
Option -md	2-26
Option -mm	2-26
Option -o filename	2-27
Option -pa	2-27
Option -pb	2-27
Option -pe	2-27
Option -pl	2-27
Option -ps	2-28
Option -pt	2-28
Option -qa	2-28
Option -qf	2-29
Option -qp	2-29
Option -qq	2-29
Option -qs	2-29
Option -qv	2-29
Option -sf	2-29
Option -sn	2-30
Option -sp	2-30
Option -sr	2-30

Option -ss	2-30
Option -su	2-31
Option -wa	2-31
Option -wd	2-31
Option -wn	2-32
Option -wl	2-32
Option -wo	2-32
Option -wp	2-32
Option -wq	2-33
Option -wr	2-33
Option -wu	2-33
Option -ww	2-34
Table Manipulation Options	2-34
Error Handling	2-34
Writing Programs for the Amiga	2-35
#pragma for Amiga Exec Calls	2-35
Calling Resident Libraries	2-36
Creating Resident Libraries	2-38
#pragma for Register Function Calls	2-39
Miscellaneous Notes on Resident Libraries	2-40
Pointer Considerations	2-41
Declare Functions That Return a Pointer	2-41
Declare Function Arguments That Are Pointers	2-42
Pointer Variables And Constants As Arguments On Calls	2-42
Other Considerations	2-43
Internal Storage Of Numeric Data	2-43
Converting Data	2-43
Additional Features	2-44
Line Continuation	2-44
String Concatenation	2-44
In-line Assembly Code	2-44

Chapter 3 - Language Specification

Compatibility Issues	3-1
Compatibility with Other Aztec Compilers	3-1
Keywords	3-2
Operators	3-2
Pre-Processing Directives	3-2
Type Information	3-2
Integers and Long Integers	3-3
Characters	3-4
Structures	3-4
Bitfields	3-5
Enum Constants	3-5
Const and Volatile	3-5
General ANSI-Compatible Features	3-6
Function Prototyping and ANSI Function-Definition	3-6
ANSI Standard	3-7
Auto Aggregate Initialization	3-10
Wide Strings and Literals	3-10
Structure Assignment and Structure Passing	3-11
Trigraphs	3-11
Escape Sequences	3-12
Long Character Constants	3-12
Register Variables	3-13
Sizeof and Size_t	3-13
Symbol Names	3-13
Preprocessor Features	3-13
Predefined Symbols	3-13

String Concatenation	3-15
Line Continuation	3-15

Chapter 4 - Assembler

Operating Instructions	4-2
EXECUTION ENVIRONMENT	4-2
Processor Support	4-2
INPUT FILE	4-2
OBJECT CODE FILE	4-3
LISTING FILE	4-3
OPTIMIZATIONS	4-3
SEARCHING FOR INCLUDE FILES	4-4
Option -i	4-4
INCLUDE Environment Variable	4-5
Include Search Order	4-5
MODULE MEMORY MODEL	4-5
Assembler Options	4-6
SUMMARY OF OPTIONS	4-6
DESCRIPTION OF OPTIONS	4-7
Option -c	4-7
Option -d	4-7
Option -o filename	4-7
Option -i	4-7
Option -l	4-7
Source Program Structure	4-8
COMMENTS	4-8
EXECUTABLE INSTRUCTIONS	4-8
LABELS	4-8
OPERATIONS	4-8

OPERANDS	4-9
DIRECTIVES	4-10
blanks	4-10
clist and noclist	4-10
cnop	4-11
cseg	4-11
dseg	4-11
dc - Define Constant	4-12
dcb - Define Constant Block	4-12
ds - Define Storage	4-13
entry	4-13
end	4-13
equ	4-13
equr	4-14
even	4-14
fail	4-14
far code	4-14
far data	4-14
freg	4-14
ifc and ifnc	4-15
ifd and ifnd	4-15
if, else, and endc	4-15
near code	4-16
near data	4-16
other ifs	4-16
include	4-16
global and bss	4-16
list and nolist	4-17
mlist and nomlist	4-17
machine	4-17
macro and endm	4-17
mc68851	4-18
mc68881	4-18

mexit	4-18
public	4-19
reg	4-19
section	4-19
set	4-19
ttl	4-20
xdef and xref	4-20
MACRO CALLS	4-20
Interfacing With C	4-21
REGISTER CONVENTIONS	4-21
ARGUMENTS TO SUBROUTINES	4-21
Variable Names	4-21
Returning from a C Function	4-22

Chapter 5 - Linker

Introduction to Linking	5-2
LINKING Hello.o	5-2
THE LINKING PROCESS	5-2
Libraries	5-3
EXAMPLE	5-4
The Order of Library Modules	5-5
Using the Linker	5-7
STARTING THE LINKER	5-7
INPUT FILES	5-8
EXECUTABLE FILES	5-8
LIBRARIES	5-9
SUMMARY OF LINKER OPTIONS	5-10
DETAILED DESCRIPTION OF OPTIONS	5-11
Option -l	5-11

Option -f	5-12
Option +l	5-12
Option -m	5-13
Option -o	5-13
Option +o[i]	5-13
Option +s, +ss, +sss	5-14
Option -t	5-14
Option -w	5-14
Option -v	5-14
Options +c and +f	5-15
Option -g	5-16

Chapter 6 - Utilities

arcv	6-2
adump	6-3
cmp	6-4
cnm	6-5
Options	6-6
Symbol Format	6-7
Symbol Types	6-7
ab	6-8
pg	6-8
dt	6-8
un	6-8
bs	6-9
Gl	6-9
du	6-10
hd	6-11

lb	6-12
Options Argument	6-12
FUNCTION CODE OPTIONS	6-12
QUALIFIER OPTIONS	6-13
The mod Argument	6-13
Reading Arguments from Another File	6-13
Basic Features of lb	6-14
HOW TO CREATE A LIBRARY	6-14
HOW TO LIST NAMES OF MODULES IN A LIBRARY	6-15
HOW MODULES GET THEIR NAMES	6-15
ORDER OF MODULES IN A LIBRARY	6-15
GETTING LB ARGUMENTS FROM A FILE	6-16
ADVANCED lb FEATURES	6-17
ADDING MODULES TO A LIBRARY	6-17
ADDING MODULES BEFORE AN EXISTING MODULE	6-18
ADDING MODULES AT THE BEGINNING OR END OF A LIBRARY	6-18
MOVING MODULES IN A LIBRARY	6-19
DELETING MODULES	6-19
REPLACING MODULES	6-20
UNIQUENESS	6-20
EXTRACTING MODULES	6-21
THE VERBOSE OPTION	6-21
SILENCE OPTION	6-22
REBUILDING A LIBRARY	6-22
DEFINING THE DEFAULT MODULE EXTENSION	6-22
HELP	6-22
ls	6-23
mkarcv	6-25
obd	6-26

ord	6-27
set	6-28
Displaying and Setting Environment Variables	6-28
setdate	6-29

Chapter 7 - Debugger

Requirements	7-1
Preview	7-1
Overview	7-2
BASIC COMMANDS	7-2
NAMES	7-3
Code and Data symbols.	7-3
Operator usage of names.	7-3
LOADING PROGRAMS AND SYMBOLS	7-3
BREAKPOINTS	7-4
MEMORY-CHANGE BREAKPOINTS	7-5
TRACE MODE	7-6
BACKTRACING	7-6
MACROS	7-6
OTHER FEATURES	7-7
Using DB	7-7
STARTING DB	7-7
TERMINATION	7-8
SUPPORT FOR SCATTER-LOADED PROGRAMS	7-8
SUPPORT FOR SEGMENTED PROGRAMS (OVERLAYS)	7-8
INPUT/OUTPUT	7-9
COMMANDS	7-9

Definitions	7-9
EXPR	7-9
TERM	7-10
ADDR	7-13
RANGE	7-13
CMDLIST	7-14
COMMAND DESCRIPTIONS	7-14
The Amiga Commands	7-14
add	7-14
adi	7-14
adl	7-14
adp	7-14
adr	7-14
ai	7-15
ak	7-15
al	7-16
aL	7-16
an	7-17
aN	7-17
am	7-17
ap	7-17
aP	7-17
aq	7-18
ar	7-19
as	7-19
aS	7-19
at	7-19
The Breakpoint Commands	7-20
bb	7-20
bw	7-20
bl	7-20
bc	7-21
bC	7-21

bd	7-21
bh	7-22
bq	7-22
br	7-23
bs	7-23
bt	7-24
bT	7-24
bu	7-24
The Clear Commands	7-25
cs	7-25
The Display Commands	7-26
db	7-26
dw	7-26
dl	7-26
d	7-26
dc	7-26
dd	7-27
dg	7-27
ds	7-27
The Go commands	7-28
g	7-28
G	7-28
The Load Commands	7-29
ls	7-29
The Memory Modification Commands	7-29
ma	7-29
mb	7-30
mw	7-30
ml	7-30
mc	7-30
mf	7-31
mm	7-31
ms	7-31

The Radix Command	7-32
n	7-32
The "Print" Command	7-32
p	7-32
A complete list of desc_codes	7-37
The Quit command	7-39
q	7-39
The Register command	7-40
r	7-40
The Single Step commands	7-40
s	7-40
S	7-40
t	7-40
T	7-40
The Unassemble commands	7-41
u	7-41
U	7-41
The Variable commands	7-42
v	7-42
V	7-42
The Macro command	7-42
x	7-42
The Display Expression Command	7-43
=	7-43
The Redirect Input/Output Commands	7-43
<	7-43
>	7-43
>>	7-43
The Help command	7-44
?	7-44
Command Summary	7-45
AMIGA COMMANDS	7-45

BREAKPOINT COMMANDS	7-45
CLEAR COMMANDS	7-46
DISPLAY COMMANDS	7-46
GO COMMANDS	7-46
LOAD COMMANDS	7-46
MEMORY MODIFICATION COMMANDS	7-46
RADIX COMMAND	7-46
FORMATTED PRINT COMMANDS	7-46
QUIT COMMAND	7-46
REGISTER COMMAND	7-47
SINGLE STEP COMMANDS	7-47
UNASSEMBLY COMMANDS	7-47
VARIABLE COMMAND	7-47
MACRO COMMAND	7-47
DISPLAY EXPRESSION	7-47
INPUT/OUTPUT COMMANDS	7-47
HELP COMMAND	7-47

Chapter 8 - Technical Information

Program Organization	8-2
PLACEMENT OF SEGMENTS OF MEMORY	8-3
SEGMENT SIZE	8-4
Segmentation and Segmented Code	8-5
SEGMENTATION	8-5
Loading a Segment	8-5
Unloading a Segment	8-6
Segload	8-6
Linking and Startup Code	8-7

Segment Construction	8-7
Segment Data Table	8-8
Hunk Data	8-8
SEGMENTED CODE	8-8
Programmer Information	8-8
Global and Static Data	8-9
Operator Information	8-9
Example	8-9
Including Modules from Libraries	8-10
Reselecting Segments	8-10
Object Module Libraries	8-11
Assembly Language Functions	8-11
CONVENTIONS FOR C CALLABLE, ASSEMBLY LANGUAGE	
FUNCTIONS	8-12
Names of External Functions and Variables	8-12
Function Calls and Returns	8-13
Global Variables	8-14
Register Usage	8-15
Embedded Assembler Source	8-15
Interrupt Service Routines	8-16
Creating SubTasks	8-17
Floating Point Information	8-18
CREATING A FLOATING POINT FORMAT PROGRAM ...	8-19
FLOATING POINT FORMATS	8-19
Building Programs that Run from Workbench	8-19
OVERVIEW	8-19
STEP BY STEP	8-20

Chapter 9 - Error Messages

List of Error Messages	9-2
Compiler Error Messages	9-2
Fatal Compiler Error Messages	9-6
Internal Errors	9-7
Assembler Error Messages	9-7
Opcode Error Messages	9-7
Directive Error Messages	9-8
Syntax Errors	9-9
Linker Error Messages	9-10
Command Line Errors:	9-10
I/O Errors	9-10
Errors in Use of Memory	9-11
Errors Arising From Source Code	9-11
Compiler Error Messages	9-12
Fatal Compiler Error Messages	9-58
Internal Errors	9-62
Assembler Error Messages	9-62
Opcode Error Messages	9-62
Directive Error Messages	9-66
Syntax Errors	9-69
Linker Error Messages	9-69
Explanation of Linker Error Messages	9-70
Command Line Errors	9-70
I/O Errors	9-71
Errors in Use of Memory	9-73
Errors Arising From Source Code	9-74

OVERVIEW

COMPILER

LANGUAGE
SPECIFICATIONS

ASSEMBLER

LINKER

OVERVIEW

1

Chapter 1 - Overview

Introduction

This is the *Aztec C Reference Manual* and is included as a part of your total Aztec C documentation package. See page 1-3 of your *Aztec C User Guide* for a listing of all of the Aztec C configurations and documentation that are available for the Amiga.

The *Aztec C User Guide* showed you how to install your software and how to begin using Aztec C easily and quickly. This *Aztec C Reference Manual* presents more advanced programs and a more thorough understanding of how the Aztec C package works.

The *Aztec C Library Manual* includes an overview, listing, and description of all the library functions that are included with your Aztec C package, and a listing of the Amiga Workbench function names and syntax.

Of course, this *Aztec C Reference Manual* also contains a detailed Table of Contents and an Index.

ANSI C

Aztec C is fully ANSI standard. If you have used Aztec C prior to Version 5.0, you should note that this standardization has caused the compiler and its options to change substantially. Before using Aztec C, you should carefully review the **Compiler** chapter.

Other chapters that have been affected by the ANSI standardization and should be reviewed include **Assembler**, **Utilities**, **Technical Information**, **Error Messages**, and **Style**.

If you are a previous user, you should also note that a new chapter has been added to this manual: **Language Specifications**. This chapter discusses Aztec C's implementation of ANSI C.

Chapter 2 - Compiler

This chapter describes how to use the Manx Aztec C Compiler. The Aztec C compiler is implemented according to the draft proposed ANSI standard. For information on the C language and its use, refer to the section at the beginning of your *Aztec C User Guide*, "Suggestions for Further Reading," which lists several tutorial texts for the C student.

This chapter contains four topics: 1) how to use the compiler, 2) compiler options, 3) error handling, and 4) programmer information, such as the use of register variables and the writing of machine-independent code.

Using the Compiler

To invoke the Aztec C compiler, use the following command:

```
cc [options] filename.c
```

where *[options]* specifies optional parameters, and *filename.c* is the name of the file containing the C source program. Options can appear either before or after the name of the C source file.

The compiler reads C source statements from *filename.c*, translates them to assembly language, and writes the results to an output file.

If the **QuikFix** option is enabled (option **-qf**) the compiler on encountering errors will dynamically invoke an editor and interactively help locate

source lines with errors. A windowed display is provided of the associated error message. This facility is described in the *Aztec C User Guide*.

When the compiler is finished, it activates the Manx assembler, unless it is told not to. The assembler translates the assembly language source into relocatable object code, writes the result to another file, and deletes the assembly language source file. Option `-a` tells the compiler not to start the assembler.

To abort a compilation, press the Control-C keys.

Input File

When you execute the compiler, the C source input file can be specified as a simple filename or as a complete path specification. The default assumes that the input file is in the current directory. For example, if the file `prog1.c` contains C source and is in the directory `dsk2:db/source`, you can compile it by using the following command:

```
cc dsk2:db/source/prog1.c
```

If the directory containing this file is also the current directory, you could compile the file with the command

```
cc prog1.c
```

And if the current directory is `db`, on the `dsk2:` volume, you could compile the file with the command

```
cc source/prog1.c
```

Source Filename Extensions

If the command that starts the compiler does not specify the extension of the file containing the C source, the compiler assumes that the extension is `.c`. For example, the command

```
cc prog
```

compiles a file named **prog.c** in the current directory.

Although **.c** is the recommended file extension name, it is not mandatory. The specification

```
cc prog.prq
```

reads the file **prog.prq** from the current directory as the input to the compiler.

The Output Files

Creating an Object File

Normally, when you compile a C program you are interested in the relocatable object code for the program, and not in its assembly language source. Because of this, the compiler by default writes the assembly language source for a C program to an intermediate file and then automatically starts the assembler. The assembler then translates the assembly language source to relocatable object code, writes this code to a file, and erases the intermediate file.

By default, the object code generated by a compiler-started assembler is sent to a file whose name is derived from that of the file containing the C source by changing its extension to **.o**. This file is placed in the directory that contains the C source file.

For example, if you started the compiler with the command:

```
cc prog.c
```

the file **prog.o** is created, containing the relocatable object file for the program. You may explicitly specify the name of the object file using the compiler option **-o**. For example, the command

```
cc -o myobj.rel prog.c
```

compiles and assembles the C source that is in the file **prog.c**, writing the object code to the file **myobj.rel**.

When the compiler is going to start the assembler automatically, by default it writes the assembly language source to the file `ctmpxxx.xxx`, where `xxx` are numbers chosen such that the file name is unique. The file is placed in the directory defined by the `CCTEMP` environment variable. If `CCTEMP` doesn't exist the file is placed in the current directory.

The `CCTEMP` environment variable can be used to pass the intermediate assembler file to the assembler through a RAM disk. If you are using floppy disks, a RAM disk can definitely save time. If you are using a hard disk, the time savings may be minimal.

If you are interested in the assembler source, but still want the compiler to start the assembler, specify option `-at` when you start the compiler. This causes the compiler to send the assembly language source to a file whose name is derived from the file containing the C source, and whose extension is set to `.asm`. The C source statements are included as comments in the assembly language source. For example, the command

```
cc -at prog.c
```

compiles and assembles `prog.c`, creating the files `prog.asm` and `prog.o`.

Creating an Assembly Language File

In some programs, you may not want the compiler to start the assembler automatically. For example, you may want to modify the assembly language generated by the compiler for a particular program. In such cases, use compiler option `-a`, which prevents the compiler from starting the assembler.

When you specify option `-a`, by default the compiler sends the assembly language source to a file whose name is derived from that of the C source file, by changing the extension to `.asm`. This file is placed in the same directory as the one that contains the C source file. For example, the command

```
cc -a prog.c
```

compiles, without assembling, the C source that is in `prog.c`, sending the assembly language source to `prog.asm`.

When using option **-a**, the **-o** option specifies the name of the file to which the assembly language source is sent. For example, the command

```
cc -a -o ram:temp.asm prog.c
```

compiles, without assembling, the C source in **prog.c**, sending the assembly language source to the file **temp.asm** on the volume named **ram:**. When option **-a** is used, sub-option **-t** causes the compiler to include the C source statements as comments in the assembly language source. Usually you will want to also specify **-at** to request **cc** to insert the original C source as comment code when it generates the assembly file. (See the "Compiler Options" section of this chapter for more information.)

Searching for #include Files

By default the Aztec C compiler searches the current directory to locate files specified in **#include** statements. It can also search a user-specified sequence of directories for such files, thus allowing program source files and header files to be contained in different directories.

Compiler option **-i** and the environment variable **INCLUDE** define the directories in which the compiler searches for **#include** files. The compiler automatically searches the current directory for a **#include** file if the following conditions are met:

- (1) the compiler is started without specifying option **-i**,
- (2) There is not an **INCLUDE** environment variable, and
- (3) the **#include** statement does not specify the drive and/or directory containing the file.

If a **#include** statement specifies either the drive or directory, only that location is searched for the file.

Option -i

Compiler option **-i** defines a single directory to be searched for the file that was specified in a **#include** statement. The path descriptor follows option

-i. A space between -i and the path descriptor is allowed but not required.

The specification

```
cc -i sys:db/include
```

directs the compiler to search the **sys:db/include** area when looking for an include file.

Multiple -i options can be specified when starting the compiler, if desired, thus defining multiple directories to be searched.

You can also specify where include files are kept, using the INCLUDE environment variable. See your *Aztec C User Guide* for more information on using the INCLUDE environment variable.

#include Search Order

When the compiler encounters a **#include** statement, it searches directories for the file specified in the statement in the following order:

- if the filename is delimited by double quotes, *"filename,"* the current directory is searched.
- if the name is delimited by angle brackets, *<filename>*, the current directory is searched only if no -i options are specified and if the INCLUDE environment variable does not exist.
- directories specified in option -i are searched, in the order listed on the line that started the compiler.
- directories specified in the INCLUDE environment variable are searched, in the order listed.

Pre-compiled #include Files

To shorten compilation time, the compiler supports pre-compiled **#include** files.

This option only applies to a block of **#include** files that are located in the beginning of a file. For example, if you place a **#undef** between two **#include** file statements, it will not have the desired effect.

To use this feature, you must first create the pre-compiled header file. This is done by compiling the desired header files with **-ho** option. This option tells the compiler not to generate any code but to create symbol table information which it stores in the file whose name was specified after the **-ho** option. This symbol table file can then be included on subsequent compiles by specifying the **-hi** option followed by the name of the symbol table file. The compiler adds into its symbol table the pre-compiled symbol table information. When the compiler encounters an **#include** statement of a header file for which it has pre-compiled symbol table information, it does not have to compile it, even if the include is nested within another include. This can save a considerable amount of time.

Since there are options that cause the compiler to make different assumptions about the size of data items, it is important to use pre-compiled header files that match the data size assumptions of the compilations where they are specified.

Option **-ho** tells the compiler to write its symbol table to a file. The name of the file follows the **-ho** and a separating space as the next argument. For example, you might create a file named **x.c** that consists only of include statements for all the header files that you want pre-compiled. You could then generate a file named **x.dmp** that contains the symbol table information for these header files by entering the following command:

```
cc -ho x.dmp x.c
```

Option **-hi** tells the compiler to read pre-compiled symbol table information from a file and uses the normal include search path. The name of the file follows the **-ho**, with one intervening space. For example, to compile the file **prog.c** that accesses the header files defined in **x.c**, and to have the compiler preload the symbol table information for these files from **x.dmp**, enter the following command:

```
cc -hi x.dmp prog.c
```

The **-hi** option actually initializes the compiler symbol table. For this reason, only one **-hi** is permitted, and any conditionally-defined objects are in-

cluded or not, depending on definitions when the **pre-compiled header** (.dmp) file was created, and not when it is used.

Memory Models

The memory model used by a program determines how the program's executable code makes references to code and data. This in turn indirectly determines the amount of code and data that the program can have, the size of the executable code, and the program's execution speed.

Before getting into the details of memory models, following is a brief description of the sections into which a C-generated program is organized. The sections of a program include:

- **code**, containing the program's executable code
- **data**, containing its global and static data
- **stack**, containing its automatic variables, control information, and temporary variables
- **heap**, an area from which buffers are dynamically allocated.

There are two attributes to a program's memory model: One attribute specifies whether the program uses the **LARGE DATA** or the **SMALL DATA** memory model and the other attribute specifies whether the program uses the **LARGE CODE** or **SMALL CODE** memory model.

Large Data versus Small Data

The fundamental difference between a large data and a small data program concerns the way that instructions access data segment data: A large data program accesses the data using position-dependent instructions, and a small data program accesses the data using position-independent instructions. An instruction makes position-dependent reference to data in the data segment by specifying the absolute address of the data; it makes a position-independent reference to data in the data segment by specifying the location as an offset from a reserved address register.

Other differences between large data and small data programs result from this fundamental difference. These other differences are:

1. There is no limit to the amount of global and static data that a large data program can have. A small data program, on the other hand, can have 64K bytes at most of global and static data.
2. For a small data program, an address register must be reserved to point into the middle of the data segment. This is the **a4** register in the Amiga.
3. For a large data program, an instruction that wants to access data in the data segment contains the absolute address of the data, and hence does not need this address register.
4. It takes more time to load a code segment for a large data program than for a small data program. The reason for this is that the absolute address of the data contained in the data segment is not known until a program is loaded. Therefore, instructions that access data segment data using absolute addresses must be adjusted when the code segment containing the instructions is loaded, whereas instructions that access data segment data in a position-independent way do not need to be adjusted.
5. A code segment is larger when its program uses large data than when it uses small data, because a reference to data in a data segment occupies a 32-bit field in a large data instruction, and occupies a 16-bit field in a small data instruction.
6. A program is slower when it uses large data than when it uses small data, because it takes more time for an instruction to access data when it specifies the absolute address of the data than when it specifies the data's offset from an address register.

Large Code Versus Small Code

The fundamental difference between a large code and a small code program concerns the way that instructions in the program refer to locations that are

located in the code segment: For a large code program the reference is made using position-dependent instructions, and for a small code program the reference is made using position-independent instructions. An instruction makes position-dependent reference to a code segment location by specifying the absolute address of the location; it makes a position-independent reference to a code segment location by specifying the location as an offset from the current program counter. (small code will be used in most cases, due to the nature of the way it works. Large code is used primarily for absolute function pointers.)

Other differences in large data and small data programs result from this fundamental difference. These other differences are:

1. The size of a code segment is unlimited for both large code and small code programs. An instruction in a large code program can directly call or jump to the location, regardless of its location in the code segment. An instruction in a small code program can only directly call or jump to locations that are within 32K bytes of the instruction.

To allow instructions in small code programs to transfer control to any location, regardless of its location in the code segment, a "jump table," which is located in the program's data segment, is used. For example, if a location to which an instruction wants to transfer control is more than 32K bytes from the instruction, the transfer is made indirectly, via the jump table: The instruction calls or jumps to an entry in the jump table, which in turn jumps to the desired location. A jump instruction in a jump table entry refers to a code segment location using an absolute, 32-bit address, and hence can directly access any location in the program's code segment.

When a small code program is linked, the linker automatically builds the jump table: If the location to which an instruction wants to transfer control is outside the instruction's range, the linker creates a jump table entry that jumps to the location and transforms the pc-relative instruction into a position-independent call or jump to the jump table entry.

2. A code segment can contain data as well as executable code. An instruction in a large code program can access data located anywhere in the code segment, because it accesses code segment data using position-dependent instructions, in which the location is referred to

using a 32-bit absolute address. An instruction in a small code program can only access code segment data that is located within 32K bytes of the instruction.

3. For a small code program to access the jump table, an address register needs to be reserved and set up to point into the middle of the program's data segment. If the program also uses small data, the same address register is used for both jump table accesses and normal accesses of data segment data. For a large code program, this address register is not needed for the referencing of locations in the code segment.

4. A program takes longer to load if it uses large code than if it uses small code. Instructions in a large code program that reference a code segment location must be adjusted when the program is loaded, since such instructions must contain the absolute address of the location and since this is not known until the program is loaded.

Instructions in a small code program that reference code segment locations need not be adjusted, since they are always independent of the location at which the code segment is loaded: If the location is within 32K of the referencing instruction, the instruction is pc-relative; and if it is outside this range, the instruction is a position-independent jump to a jump table entry.

When a small code program that contains a jump table is loaded, its jump table entries must be adjusted, since these are jump instructions to code segment locations, where each instruction must contain the absolute address of the destination address. However, it should take less time to adjust the jump table for a small code program than to adjust the code segment of a large code version of the same program, since for any destination of a jump or call instruction a small code version of a program will have at most one jump table entry needing adjustment, whereas a large code version of the program may have many jump or call instructions to the same location that need to be adjusted.

5. A code segment is larger when its program uses large code than when it uses small code, because instructions that reference code segment locations by specifying an absolute address use a 32-bit field

to define the location, whereas instructions that reference data by specifying a pc-relative address or an offset from an index register use a 16-bit field to define the location.

6. A program is usually slower when it uses large code than when it uses small code, because it takes more time for an instruction to reference a code segment location when it specifies the absolute address of the data than when it specifies the location in a pc-relative form.

A large small code program that has lots of indirect transfers of control via the jump table may not differ much in execution time from a large code version of the same program, since the small code indirect transfer via the jump table will take more time than the large code direct transfer.

Selecting a Module's Memory Model

You define the memory model to be used by a module when you compile the module, by specifying or not specifying the following options:

- mc Module uses large code. If this option is not specified, the module will use small code.
- md Module uses large data. If this option is not specified, the module uses small data.

For example, the following commands compile `prog.c` to use different memory models:

<code>cc prog</code>	small code, small data
<code>cc -mc prog</code>	large code, small data
<code>cc -md prog</code>	small code, large data
<code>cc -mcd prog</code>	large code, large data

Libraries

The modules in the Aztec C libraries use the small code, small data memory model. Most libraries have large code and large data equivalents, e.g., `cl.lib` is the large data version of `c.lib` and may be linked in with `-lcl` instead of `-lc`.

Unless you have extraordinary requirements, you will probably never need to use `-mc` on the Amiga. If one of your functions calls another function in the same segment that is more than 64K bytes away, the linker automatically forces the target routine into the application's jump table and directs the call through the jump table. See the *Amiga ROM Kernel Reference Manual* for a description of the jump table. Function calls across segment boundaries always go through the jump table.

Furthermore, your first code segment may be of arbitrary size, again without the use of `-mc` or any other special action on your part. The application runtime startup code (`crt0.o`) takes care of fixing up the jump table, using information supplied by the linker.

When you compile a module with `-md`, all references in that module to global and static data are set up as 4-byte absolute values. The use of `-md` is never necessary until the size of the initialized and uninitialized data segments (see the output from the linker) exceed 64K bytes. Large-data references, Data-to-data references or code-to-data references, are relocated by the DOS loader.

Multi-module Programs

The modules that you link together to form an executable program can use different memory models, with the following caveat.

When large data and small data modules are linked together, the linker creates an arbitrarily large data segment, without attempting to sort the data into those that are accessed by large data modules and those that are accessed by small data modules.

When the program is running, address register `a4`, which the small data modules use to access data, points 32K from the beginning of this data segment.

Here is the caveat: data that the small data modules attempt to access must be within 32K bytes of the location pointed at by `a4`. The linker detects data accesses by small data modules for which this condition is not satisfied, and issues a message. If you get this message, try reordering the way in which the linker encounters them. If that does not solve the problem, you will have to recompile the small data modules, to make them use large data.

Code Segmentation

By default, the linker creates a program that has a single code segment. As you have seen, there is no limit on the size of this segment, regardless of the memory model used by the program.

However, there is a limit on the amount of memory in a machine. Also, the larger a program's code segment, the worse its performance. If the program uses small code, its execution speed degrades as the program gets larger, because more transfers of control will have to be done indirectly, via the jump table. And if it uses large code, it will be larger and slower than a small code version of the same program.

The Aztec linker allows you to divide a program's code into multiple segments. When the program is running, code segments are brought into memory as needed. When a code segment is no longer needed in memory, the memory it occupied can be released and reused, either for another code segment or for some other purpose.

One advantage of partitioning a large program into code segments is that it allows a program to be larger than available memory. Another advantage is that it allows the code to use small code memory model without the degradation caused by indirect transfers of control via the jump table.

There is some loss of performance in a segmented program, since the segments may be loaded into memory from disk more than once to be executed. With careful partitioning of the program, the loss can be minimized.

Compiler Options

Option Philosophy

Most of the compiler options are set up as toggles, which means that they can be either on or off. Most options default to off. The defaults can be changed by creating an environment variable, CCOPTS. Options specified directly to the compile command will override options specified through the CCOPTS environment variable.

With a few exceptions, options are grouped around a common function. The first letter of an option identifies the group. The group letters are:

- a** Assembly language output control
- b** Debugging control
- c** Chip or processor control
- f** Floating point control
- h** pre-compiled header file control
- m** Memory model control
- p** Parser control
- q** Output control
- r** Register control
- s** Optimization control
- w** Warning control

After the group letter, one or more individual options may be specified. If an individual option letter occurs and is not preceded by a **0** (zero), the associated option is turned on. Multiple individual options can be specified.

To turn an option off, the character **0** (zero) must appear after the group letter and before the options to be turned off. **-p0t**, for instance, turns off trigraphs and **-pt** turns them on.

Combinations of options can be used to produce very specific results. To enable full ANSI syntax checking with the singular exception of trigraphs, for example, you would use the option **-pa0t**. The **a** option of the **p** group specifies full ANSI which includes trigraphs. The **0t** option turns trigraphs off. Since options are scanned left to right the combination **-pa0t** produces the desired result. **-p0ta** would not produce the intended result. Since the **a** option is scanned after the **0t** option, the **0t** option is cancelled.

CCOPTS Environment Variable

You can override the default settings of the compiler by using the environment variable CCOPTS.

If you want to use 16 bit ints as the default, for example, you could say

```
set CCOPTS=-ps
```

which would cause the compiler to use 16 bit ints instead of the standard default of 32 bits.

The CCOPTS specification should have no embedded blanks including both sides of the equal (=) sign. If blanks are used as separators the whole specification must be enclosed in quotes. Thus:

```
set CCOPTS=-ps
```

or

```
set "CCOPTS = -ps"
```

but not

```
set CCOPTS = -ps
```

Options passed directly to the compiler override the CCOPTS environment variable. If the CCOPTS environment variable was set to **-ps** and you specified **-p0s** as a direct option to the compiler, then the CCOPTS **-ps** option would be overridden.

If you wish to specify more than one option with the CCOPTS environment variable, then each option group must be separated by a blank and the whole string enclosed in quotes. For example,

```
set "CCOPTS=-ps -mcd -wo"
```

would set the **-ps**, **-mc**, **-md**, and **-wo** options. Note that CCOPTS must be specified in upper case.

Options that require additional arguments must not be specified using the CCOPT environment variable. This includes the **-o**, **-d**, **-i**, and **-h** options.

Option List

- 3 Interpret options that follow according to version 3.6 rules
- 5 Interpret options that follow according to version 5.0 rules
- a Causes the compiler not to start the assembler after it has compiled a program. Only an assembly file is created.
- at Same as -a, but also imbeds C source statements into the assembly code.
- bd Enables stack depth checking.
- bs Produces sdb debugging information
- c2 Compiler will create 68020 code.
- d Defines a symbol for the preprocessor.
- fa Generates code for Amiga IEEE floating point.
- ff Generates code for Motorola Fast Floating point.
- fm Generates code for Manx IEEE. Defaults to on.
- f8 Generates code for 68881 Floating Point format.
- hi Reads pre-compiled header files into its symbol table.
- ho Writes compiler symbol table. Used for making pre-compiled header files.
- i Specifies path for include files.
- k Compile code according to the K&R (UNIX version 7) standard
- ma Forces alignment of short and long data items. Defaults to on.

- mb Generates the public `.begin` statement. Defaults to on; turn off for drivers.
- mc Generates code that uses large code memory model.
- md Generates code that uses large data memory model.
- me Aligns strings on word boundaries. This forces all strings to start on an even address.
- mm Puts data in code segment.
- ms Puts static strings in data segment.
- o Specifies name to be used for output file.
- pa Turns on ANSI preprocessor and trigraphs. Turns off 'cdemou' suboptions for -p.
- pb make bitfield unsigned by default
- pc Allows extra characters after a `#endif` or `#else`. Defaults to on.
- pe Causes enums to occupy only the amount of space needed. Defaults to off.
- p1 Defines integers to be 32 bits instead of 16 bits. Defaults to on.
- po Uses old (3.6) preprocessor.
- pp make characters unsigned by default
- ps Defines integers to be 16 bits instead of 32 bits.
- pt Looks for trigraphs in the input stream. Defaults to off.
- pu Uses unsigned preserving rules.

- qa Causes generated prototypes to use `__PARMS()` syntax.
- qf Enables QuikFix.
- qp Generates prototypes for all non-static functions defined within the current file.
- qq Disables the startup and error messages from being displayed
- qs Generates prototypes for all static functions defined within the current file.
- qv Generates verbose information on memory usage.
- r4 Uses `a4` for register variables.
- sa Enables two pass assembly for branch squeezing and other optimizations.
- sb Enables use of built-in in-line functions for the string operations `strcpy()`, `strcmp()`, and `strlen()`.
- sf Generates an optimized `for(;;)` loop that places the test at the bottom of the loop.
- sm Defines the `__C_MACROS_` macro to replace some functions with in-line macro expansions defined in the header files.
- sn If no local variables are created on the stack for a function, does not generate the `LINK` and `UNLK` instructions.
- so A shorthand way to specify the suboption 'afmnprs' for -s.
- sp Delays the popping of arguments until necessary.
- sr Allocates registers based on weighted usage counts.

- ss** Finds duplicate strings and replaces them with a pointer to the first occurrence.
- su** Allocates registers based on weighted usage counts, but allocates to user specified variables first.
- wa** Complains on arguments which do not match the prototype specification.
- wd** Generates warnings for old style K&R definitions.
- we** Quits on warnings. Treats warnings as errors.
- wl** Shorthand for **-waru** and stands for lint.
- wn** Do not generate warnings on direct pointer to pointer conversions
- wo** Causes pointer/int conflicts to generate warnings rather than errors.
- wp** Generates a warning if a function is called without a prototype being defined for a function.
- wq** Prints warnings to the file **aztecC.err**.
- wr** Warns if function return type does not match declared type.
- ws** Ignores all warnings.
- wu** Warns about unused local variables.
- ww** Allows compiler to continue beyond 5 errors.

Option Descriptions

Option -3

The option(s) following this option will be interpreted according to version 3.6a syntax. The compiler option specification scheme for version 5 was reorganized to make it more coherent and to allow for future requirements. The -3 option is provided to reduce the impact of conversion on version 3.6a users. The document, *The Transition from Aztec C version 3 To Aztec C version 5 On The Commodore Amiga*, should be consulted. The following is a simple example of the use of this option,

```
cc -3 +x3 +c +d prog.c
```

There are several fine points to using the -3 option.

- ☐ The default integer size in version 5 is 32 bits. The default in the version 3 compiler and earlier versions was 16 bits. The -3 option sets the default integer size to 16 bits to maintain compatibility. If 32 bit integers are required, then specify the +p or +l options following the -3 option.
- ☐ If the -3 option is used, options using version 5 syntax can be specified before the -3 option or following the -5 option.
- ☐ The -k option is a special case and can be used anywhere in the option specification list.
- ☐ The version 3.6a option +x1 is no longer supported.
- ☐ The version 3.6a table size options, (-e, -z, and the like) are accepted but have no effect. The version 5 compiler allocates its internal memory space dynamically.
- ☐ The version 3 +e option is accepted but has no effect. This option is no longer necessary since version 5 always preserves the registers that were preserved by this option.

Setting the CCOPTS environment variable to -3k is a simple effective way to delay changing makefiles or other command files that invoke the Aztec C compiler using the old style option syntax. The command line looks as follows,

set CCOPTS=-3k

The **-3** option is not a global version 3.6a compatibility switch. For a discussion of library routines, library names, ANSI syntax changes and other version 3.6a to version 5 differences it is strongly advised that you consult the transition document cited earlier.

Option -5

The option(s) following this option will be interpreted according to version 5.0 syntax. This option is used with the **-3** option to allow the specification of options using version 5.0 syntax after the specification of options using version 3.6a syntax. For example:

```
cc -3 +p -D GRAPHICS -e1000 -5 -sn prog.c
```

Option -c2

Directly generates 68020 instructions that will take advantage of the 68020 and 68030 chips. These instructions will not run on a 68000 or 68010.

Option -d *macro*

Option **-d** defines a symbol in the same way as the preprocessor directive, **#define**. Its usage is as follows:

```
cc -dmacro[=text] filename
```

For example,

```
cc -d MAXLEN=1000 prog.c
```

is equivalent to inserting the following line at the beginning of the program:

```
#define MAXLEN 1000
```

The separating space following the **-d** is optional. The following formats are equivalent:

```
-d MAXLEN=1000
```

```
-dMAXLEN=1000
```

Since option **-d** causes a symbol to be defined for the preprocessor, it can be used in conjunction with the preprocessor directive, **#ifdef**, to selectively

include code in a compilation. A common example is code such as the following:

```
#ifdef DEBUG
printf("value: %d\n", i);
#endif
```

This debugging code would be included in the compiled source by the following command:

```
cc -d DEBUG program.c
```

When no substitution text is specified, the symbol is defined as the numerical value one.

This capability is useful when small pieces of code must be altered for different operating environments. Rather than maintaining two copies of such a program, this compile time switch can be used to generate the code needed for a specific environment. For example,

```
#ifdef AMIGA
amigainit();
#else
ibminit();
#endif
```

Option -fa

Use this option when linking with the **ma.lib** library. Doubles and long doubles are represented as 64 bit IEEE format and floats use 32 bit IEEE. For support for long doubles, use the **-fm** option listed below.

Option -ff

Use this option when linking with the fast floating point **mf.lib** library. Float and double are represented as 32 bit values in this library and long doubles as 64 bit values.

Option -fm

This is the default math option. Use it when linking with the Manx IEEE **m.lib** library. It supports the long double type as 96 bit format, double as 64 bits, and float as 32 bit.

Option -f8

Use this option when linking with the **m8.lib** library. It generates code to directly use the 68881 floating point chip.

Option -hi *filename*

This option specifies the name of an input file containing pre-compiled header file information. The input file is created using the **-ho** option.

Separating space between the identifier **-hi** and the filename is optional.

Option -ho *filename*

This option specifies the name of an output file for pre-compiled header information. This information is later used with the **-hi** option to compile programs that use the same header files. The use of pre-compiled header files can significantly shorten compile times.

If you are including large header files, like **functions.h**, in your programs you should use pre-compiled header files.

Separating space between the identifier **-ho** and the filename is optional.

Only one pre-compiled header file may be used per compilation, but it may contain information from numerous header files. For example, if the C source files for a program included the header files **stdio.h**, **fcntl.h**, **math.h**, and **functions.h**, you could speed up the compilation by precompiling these files. To do that, create a file named **x.c** that consists of the following lines:

```
#include <stdio.h>
#include <fcntl.h>
#include <functions.h>
#include <math.h>
```


Create the pre-compiled header file **x.dmp** using the following command line:

```
cc -ho x.dmp x.c
```

Then, when compiling a source file that includes any of these header files, specify **-hi x.dmp**. If the source file was named **prog.c** then the command line would look as follows:

```
cc -hi x.dmp prog.c
```

The use of pre-compiled header files does not require any changes to your source files. The **#include** statements remain without change. The only difference between programs compiled with and without pre-compiled headers should be the **-ho** specification.

The list of header files referenced in your source programs and those contained in the pre-compiled header file do not have to perfectly match. The source program can reference header files that are not pre-compiled and the pre-compiled header file can contain headers that are not referenced in the source program. The compiler will extract header information from the pre-compiled header file and then generate any additional header information that is needed. *

The use of pre-compiled header files may seem a bit complicated at first, but for programs that include large header (**.h**) files the time savings is worth the effort of learning how to use them.

Option -I *path*

Option **-i** causes the compiler to search in a specified area for files included in the source code.

The name of the area follows the **-i** with an optional separating space. For example, the following defines directory **source/inc** on volume **sys**: search area:

```
-i sys:source/inc
```

Option -k

The **-k** option causes the compiler to adhere to K&R (UNIX version 7) C syntax, rather than ANSI. This option is useful in compiling code written

in Aztec C version 3.6a or some other K&R conforming compiler. The **-k** option is equivalent to compiling with the options **-pou0a -wo**.

Option **-mb**

By default the compiler generates:

```
public .begin
```

The compiler generates the reference to **.begin** to force loading of the program startup code that eventually calls **_main()** from the library. If you are writing a driver or other piece of stand-alone code, then you do not want the standard program startup. So, to avoid getting the standard startup, you can either define your own **.begin** symbol, or compile all files with **-m0b** to prevent the **.begin** statement from being generated. All libraries are compiled with **-m0b**.

Option **-mc**

Causes the compiler to define the name **_LARGE_CODE**, forcing the assembler to generate 32 bit absolute code references rather than **a4** relative jump table references.

Option **-md**

Causes the compiler to define the name **_LARGE_DATA**, forcing the assembler to generate 32 bit absolute data references rather than **a4** relative references. This memory model is used in some of the header files to switch between an external definition of some hardware addresses and a macro definition with the address hard-coded in.

Option **-mm**

This suppresses all **dsegs** from being generated by the compiler, which causes code and data to be intermixed. This might be useful for some ROM applications and has been included here for compatibility with Greenhills C.

Option -o filename

This option is used to override the default output filename. The separating space between the identifier **-o** and filename is optional. A detailed discussion of compiler output files appears earlier in this chapter.

Option -pa

This switch is used to turn on all the ANSI checking and turn off any special extensions. A program which compiles with the **-pa** flag should compile with any other ANSI standard C compiler.

Option -pb

This option causes bit fields to be treated as unsigned. The version 5 default is signed. For example

```
int pFlags :3;
```

by default defines a bit field whose value ranges from -4 to +3. If the **-pb** option is specified, the range is from 0 to +7.

Option -pe

By default the size of **enums** is the same as that of an **int**. This option places them in the smallest space that will contain them. For example, the definition:

```
enum colors {blue, red, yellow, white};
```

would use a single byte to represent **enums** of this type if the **-pe** option was used. The definition:

```
enum countries {USA, England, France=5, Germany}
```

would use a two byte word to represent **enums** of this type when using the **-pe** option.

Option -pl

Integers are represented as 32 bit format. This is the default setting and would be used in conjunction with the **c.lib** library.

Option -ps

Integers are represented as 16 bit format. With this setting, care needs to be taken in calling Amiga library routines which expect 32 bit integers or passing integer values to pointers which expect 32 bit values. Use the `c16.lib` library when using this setting.

Option -pt

Trigraphs are an attempt by the ANSI committee to support foreign keyboards and printers. Certain characters are represented as two question marks followed by another character which indicates the true character intended. For example, `??=` is equivalent to `#` and `??(` is equivalent to `[`. In most instances, this simply slows down the parser, but it must be supported for ANSI compliance. The `-pa` turns on trigraph checking, but this option is included to turn off trigraphs while retaining the rest of the ANSI compliance.

Option -qa

The `-qa` argument controls how the prototypes are generated. If `-qa` is specified, then a prototype is generated as:

```
int func _PARMS((int x, int y));
```

instead of the default:

```
int func(int x, int y);
```

The first form is used for maximum portability to pre-ANSI compilers. The following sequence is usually placed at the beginning of a header file containing these style prototypes:

```
#if _ANSI_C
#define _PARMS(x) x
#else
#define _PARMS(x) ()
#endif
```

Then, if `_ANSI_C` is defined, the macro will change the above prototype to:

```
int func (int x, int y);
```

Otherwise if `_ANSI_C` is not defined, the prototype becomes:

```
int func ();
```

which is the standard declaration of an external function defining the type returned by the function.

Option -qf

The `-qf` option enables QuikFix which is described in the *Aztec C User Guide*.

Option -qp

The `-qp` and `-qs` options control whether static functions should have a prototype generated or not. Both options may be selected together to get all the functions. However, generally, all the global definitions are combined into one header file included by all files, while static function prototypes are placed in each individual file. See also `-qa`

Option -qq

The `-qq` option disables the startup and error count messages from being displayed on the screen.

Option -qs

See the `-qp` option.

Option -qv

The `-qv` or verbose option displays information on memory usage to the screen.

Option -sf

Versions prior to 5.0 of the Aztec C compiler always generated `for(;;)` loops with the tests and increment at the bottom of the loop since this is smaller and faster. However, if you had a statement of the type:

```
for (i+0;i<n;i++)  
    do_something;
```

then **sdb** would treat the increment and the test as part of the statement since there is no **}** on which to hang it. As a result, if you tried to step through the loop, the first **s** or **t** command would cause the entire loop to be executed.

Now, the compiler defaults to using the previous style **for(;;)** loop generation where the test and increment are at the top of the loop and are thus associated with the **for(;;)** statement which is better for **sdb**. The **-sf** option, however, generates faster smaller code.

Option -sn

Normally, when a function is called, it creates a stack frame using the **LINK** instruction to save register **a5**, point it at the old **a5**, and decrement the stack pointer by the amount of local storage needed. Then all references to local variables and arguments are made through **a5**. If the **-sn** option is selected, and there is no local storage needed by the function, then the **LINK** and accompanying **UNLK** instructions are not needed and are eliminated. Arguments are accessed using the stack pointer directly.

(Note: This confuses **sdb** and causes it to run more slowly.)

Option -sp

This option saves some small amount of execution time when several functions are called one after another. This is accomplished by delaying stack operations until necessary and thereby eliminating unnecessary stack operations. The stack is corrected whenever it is necessary or if the number of bytes delayed exceeds 50 bytes.

Option -sr

-sr causes the compiler to pick which local variables should be assigned to register variables. This option can produce significant savings in code size and execution speed. This option works in conjunction with the **-sn** option since no local storage is required if all variables end up in registers.

Option -ss

This option finds duplicate strings and replaces them with a pointer to the first occurrence. This also works for common tail subsets of strings. For

example, if the two strings "highway" and "way" are used in a function, the code generated will use a pointer to the fifth letter of highway for the way string. If the string "high" also occurred, it would be stored separately, since it would not match.

Option -su

This is the same as the **-sr** option, except that **-sr** ignores any register statements when picking registers, while **-su** uses these first and then allocates any remaining registers.

Option -wa

Normally, if you call a function in the presence of a prototype, and the type of a function parameter does not match the type specified in the prototype, the type is coerced as if by assignment. Thus if you pass an **int** to a function expecting a **long**, the **int** is quietly cast to a **long**. This option causes the compiler to generate a warning if a quiet cast is generated. This is useful, since it allows you to change the code to an explicit cast which is portable to pre-ANSI compilers.

Option -wd

This option generates a warning if a function is defined using the old pre-ANSI style definition of the form:

```
int function  (a, b)  
int a, b;  
{
```

instead of:

```
int function  (int a, int b)  
{
```

This a useful tool for "ANSI-izing" your programs.

Option -wn

The **-wn** option suppresses warning messages for direct pointer to pointer conversions which do not contain an explicit cast, such as the following code fragment:

```
int *iptr;
char *cptr;
cptr = iptr;
```

Because the ANSI Standard defines that such direct conversions are illegal, the **-pa** (full ANSI) option will always generate an error for the above code. To eliminate the error when **-pa** is specified, the above could be changed to:

```
int *iptr;
char *cptr;
cptr = (char*)iptr;
```

Option -wl

The **-wl** option is a short-hand way of specifying the **-wa**, **-wr**, and **-wu** options. It is used to force the compiler to do maximum type checking.

Option -wo

Under ANSI C, pointer-integer conversions are illegal, and are usually a problem when 16 bit ints are used. The **-wo** option changes pointer-integer conversion errors into warnings, and has been included for compatibility with version 3.6a

Option -wp

This is useful for determining if a header file is not being included. For example, if you use **strlen()** without including **string.h**, then **-wp** will generate a warning.

Option -wq

By default the compiler displays error messages on the screen. This option sends them to a file named **AztecC.err**. This file is used by the QuikFix facility.

Option -wr

Normally, a function is declared without a type, which implies that it returns an **int**. However, if there are any return statements, explicit or implicit, that do not return a value or that return a value whose type disagrees with the type of the function, then a warning is generated for the return statements. Thus, the function:

```
foo()  
{  
    printf("hello\n");  
}
```

would generate a warning since it should have been declared:

```
void foo()
```

since there is no value returned. The warning would be on the `}`, which is an implicit return statement.

Option -wu

Since the compiler has the entire tree of the function in memory, it can detect that a variable has been declared but has not been used. This option directs the compiler to check for such variables and generate a warning for each that is not used. For example, the function:

```
void foo()  
{  
    int j;  
    printf("hello\n");  
}
```

would generate a warning that `j` is not used in the function.

Option -ww

Normally the compiler will pause after encountering five errors and ask if compilation should continue. The **-ww** option indicates that the compiler should not pause and should continue to the end of compilation.

Table Manipulation Options

The compiler contains several memory-resident tables in which to store information about a program it is compiling.

Note to previous users: Prior to Version 5.0, these tables were static in size with command line options being used to alter the default sizes. The compiler now dynamically allocates and frees space for all of its internal data and tables. Therefore all previous options which dealt with table sizes have been eliminated.

Error Handling

There are two types of compiler errors—fatal and nonfatal. Fatal errors cause the compiler to make a final statement and stop. Running out of memory and finding no input are examples of fatal errors. The **Error Messages** chapter describes all these errors in detail. In addition, the nonfatal errors are also discussed there.

The compiler reports any errors it finds in the source file, displaying the source line in which the error was detected, with the cursor highlighting the approximate point where the error occurred.

The compiler then displays a line containing the following information:

1. the name of the source file containing the line
2. the number of the line within the file
3. an error code
4. a message describing the error
5. the symbol that caused the error, when appropriate.

The compiler sends error messages to its standard output device. This can be redirected to a file in the normal way. Without the redirection of its

standard output, the compiler sends error messages to the console. For example, to compile `prog.c` and send error messages to the file `prog.err`, use the following command:

```
cc > prog.err prog
```

When the compiler sends error messages to the screen, it pauses after five error messages are displayed, and asks if you want it to continue. If you type `Y`, followed by a `<CR>`, the compiler continues.

If you type anything else, followed by a `<CR>`, the compiler stops.

When the compiler sends error messages to a device or file other than the screen, it processes the entire file without giving the operator the opportunity to abort the compilation, even if it detects errors.

The compiler is not always able to give a precise description of an error. Usually, it must proceed to the next item in the file to ascertain that an error was encountered. Once an error is found, it is not obvious how to interpret the subsequent code, since the compiler cannot second-guess the programmer's intentions. This may cause it to flag perfectly good syntax as an error. If errors arise at compile time, you should first correct the first error, since this may clear up some of the errors that follow.

The best way to attack an error is first to look up the meaning of the error code in the **Error Messages** chapter. You will find hints there as to what the problem might be. And you will find it easier to understand the error and the message if you know why the compiler produced that particular code. The error codes indicate what the compiler was doing when the error was found.

Writing Programs for the Amiga

#pragma for Amiga Exec Calls

Normally, when a function is called in a program, the code for that function is physically made a part of the program by the linker. This is true for all the Manx supplied functions that are in the library we provide. On the Amiga, there are a number of additional libraries which are used to interface the program with the operating system and the graphics environment.

Many of these libraries are located in ROM, although some are loaded from disk into RAM. Since these routines cannot be loaded into the program, and since the physical address of each routine can be different from one version of the ROM to another, a dynamic system of calling ROM library routines exists on the Amiga.

The structure and use of resident libraries is discussed in the Amiga Rom Kernal Manuals. In brief, each library has a pointer to a library structure. Preceding this structure is a jump table with entry points for each function in the library. When a library is opened, the pointer to the library structure is returned. When a call is made to the library, parameters are placed in the appropriate registers, the library pointer is placed in register a6 and the appropriate negative offset is called.

In versions of Aztec C prior to 5.0, when a ROM library routine was called, the arguments were pushed on the stack and a small assembly language stub was added to the program which pulled the arguments off the stack into the appropriate registers, loaded the library base pointer into a6, and called the routine. This resulted in a number of small glue routines which added overhead both in terms of size and speed.

The functions pointed to by the library vector would also be an assembly language stub that would save registers, set up appropriate registers, push the register parameters onto the stack, and then call the function which would handle the request.

Aztec C now also allows the compiler to directly generate the call to the ROM library by generating code to load arguments directly into registers, load the beginning address into a6 and call the routine. This function is implemented using a **PRAGMA**.

Pragmas are the ANSI committee's way of allowing extensions to the C language in a controlled manner. Essentially, any line beginning with **#pragma identifier** is parsed by the compiler to see if *identifier* matches any that the compiler knows about. If not, the line is ignored.

Calling Resident Libraries

Making a call to a resident library is very simple with version 5.0 of Aztec C through the use of an **amicall #pragma** statement. (**#pragma** statements

are the ANSI supported way of doing magic things with the compiler.) An **amicall** statement takes the form:

```
#pragma amicall (LibBase, offset, func(reg1, reg2, ..., regn))
```

In this statement, the *LibBase* parameter is the name of the variable which contains a pointer to the library as returned by **OpenLibrary()**. *offset* is the positive offset value for this function. *func* is the actual name of the function being called. *reg1* through *regn* are the registers in which the parameters are passed. For example, the Exec call **AllocMem()** has a pragma defined as follows:

```
#pragma amicall(SysBase, 0xc6, AllocMem(d0,d1))
```

which declares that *SysBase* is the variable containing the pointer to Exec's library base structure. The routine to be called is at offset - (0xc6) from the *SysBase* pointer. The name of the routine is **AllocMem()** and the two arguments are to be passed in registers **d0** and **d1** respectively.

Now, when the compiler sees a statement like:

```
ptr=AllocMem(sizeof(struct Node), \
MEMF_PUBLIC|MEMF_CLEAR);
```

some form of the following code is generated:

```
move.l    a6, -(sp)
move.l    #14, d0
move.l    #$1000, d1
move.l    _SysBase, a6
jsr       -$c6(a6)
move.l    (sp)+, a6
move.l    d0, _ptr
```

Thus, no assembly language glue routine is required, since the compiler handles it all automatically.

For the resident libraries that are a basic part of the Amiga computer, ANSI style prototypes and **amicall** pragmas are defined in the file **functions.h** which is found in the standard Aztec C include directory. Thus, any file which contains a reference to an Amiga library call should:

```
#include <functions.h>
```

Since there are over 500 Amiga functions, pre-compiling **functions.h** is greatly recommended (see the compiler **-hi** and **-ho** options).

So when you are creating your own resident library, the only thing you need to do to allow Aztec C programs to make calls to your library is to define the appropriate **#pragma amicall** statements in a header file and the compiler will do the rest of the work.

Creating Resident Libraries

Creating resident libraries is now very simple.

The distribution disks contain two files that can be modified to create your own resident library. The files are **libstart.asm** and **libsup.c**. These files along with file **lib.c** are a simple example of the components of a resident library.

The **libstart.asm** file contains the assembly language rom tag structure and the assembly language initialization required for Aztec C's small model. To use this routine in creating your own resident libraries, the only thing that needs to be modified is the version number and priority. To modify them reference the macros **MYVERSION** and **MYPRI** at the beginning of the file.

The **libsup.c** file contains data structures and four routines that do initialization and support. The first data structure is **libfunctab**. The first three entries in this structure are mandatory resident library entries and point to routines found in **libsup.c**. The fourth entry is reserved. The following entries are pointers to the user defined library routines. The last entry is the end of table code which is a negative one (-1). The sample routines **func1** and **func2** that are coded in this sample should, of course, be replaced with names of your own routines.

The **myInitTab** structure does not need to be changed. It points to the **libfunctab** structure and to the startup code in **libstart.asm**.

Next you will find data definitions that give the revision level, the name of the library, and an 'id' string. The library name should be set to the name of the library you are creating. The other names, **myname** and **myid**, should

not be changed unless the `libstart.asm` file is modified to mirror the changes made.

The first routine, `_main`, is called by the startup code when the library is first loaded into memory and only then. Once the library is opened, additional calls to `OpenLibrary` will call `myOpen` which will increment the library use count by one. Each time `myClose` is called, the library use count is decremented by one.

The `myExpunge` routine is called when the system requires more memory. If the library use count is zero, the resident library is removed and the associated memory is freed. If the use count is not zero, then a flag is set so that the library will be automatically removed when the open count drops to zero. These routines do not have to be changed although they can be changed.

The example has two routines `Func1` and `Func2` in the file `mylib.c`. The `Func1` routine stores the number passed to it and the `Func2` routine adds the number passed to it to the number saved by the `Func1` routine. Note that the only thing special about these routines is the declaration of an uninitialized variable matching the name of the library base. The compiler will automatically assign this variable to the register `a6` which contains a pointer to the library base when this routine is called.

The parameters to the function are actually in registers when the function is called. However, the compiler will generate code that will correctly deal with the parameters because of the `amicall` pragma for each function which is defined in the `mylib.h` file.

The file, `test.c`, contains an example that uses the resident library just described. Note that the file `mylib.h` is included in `test.c` and it specifies which register to use for each parameter.

#pragma for Register Function Calls

It is possible to use registers to pass arguments to regular functions. To do this use the `regcall` pragma which is defined similar to the `amicall` pragma.

```
#pragma regcall([return=]funcname (arg1, arg2,...,argn))
```

This is interpreted in the same manner. This pragma will affect both the call to the function as well as the function itself which will look in the specified registers for arguments.

Miscellaneous Notes on Resident Libraries

- Any function referenced in an **amicall** or **regcall** pragma must have been previously declared (preferably with a prototype of the arguments). For example:

```
#pragma      amicall(mylibBase, 0x1e, func1(d0))
```

by itself will generate an error, while:

```
void func1(long value);  
#pragma amicall(mylibBase, 0x1e, func1(d0))
```

is correct.

- The pragmas, **libcall** and **syscall**, are parsed and handled just like **amicall**.
- If the compiler is invoked with small code or data, each function with an **amicall** pragma saves **a4** on entry and loads it with a pointer to the libraries data. On exit **a4** is restored. If invoked with large code and large data, **a4** is not affected.
- The file **mylib.fd** is included on the distribution disk as an example of a program definition file for the MAPFD program. The file specifies parameter passing conventions and function names. The MAPFD program then produces a set of #pragmas and assembly language glue routines. You can follow this as a guideline for process your own libraries with MAPFD. The .fd file you create should be saved for possible use with an MAPFD program for BASIC or some other language.

Pointer Considerations

Pointers are 32 bits wide in Aztec C compiled programs, whereas `ints` may be 16 or 32 bits wide. Because of this difference, a program whose `ints` are 16 bits wide will not work if the program assumes that pointers and `ints` are the same size, and that pointers and `ints` are treated the same. Since by default an `int` is 32 bits wide, this is only a problem when compiling with the `-ps` option.

It is easy for a C program to accidentally or purposely make these assumptions, since in C the type of an undeclared function or function argument is assumed to be `int`. A program making these assumptions runs on machines for which the assumptions are true, of course. But when you use Aztec C to recompile the program, it will malfunction.

A program that uses pointers should obey the following rules:

- If a function returns a pointer, explicitly declare it, both in the function itself and in any module that calls the function.
- If a function argument is a pointer, explicitly declare it in the function itself, and be careful not to accidentally pass an `int` as the argument.

The following paragraphs discuss these rules in detail.

Declare Functions That Return a Pointer

The following code demonstrates the importance of declaring a function that returns a pointer, both in the function itself and in a function that calls it.

For this example, assume that the function `g` must be passed a pointer.

```
char *f();  
...  
g(f());
```

The compiled code passes the 32-bit pointer returned by the function `f` to the function `g`. If `f` was not declared as returning a pointer, the compiled

code, when using the **-ps** option, assumes that a 16-bit **int** was returned by **f**, and hence passes only part of the pointer returned by **f** to **g**.

Declare Function Arguments That Are Pointers

The following code demonstrates the importance of declaring that a function argument is a pointer in the function itself.

For this example, assume again that the function **g** must be passed a pointer.

```
f(y)
char *y;
{
    ...
    g(y);
    ...
}
```

The compiled code takes the 32-bit pointer **y**, that is passed to the function **f**, and passes it to the function **g**. If the declaration **char *y** is omitted, the compiler, when using the **-ps** option, assumes that **y** is a 16-bit **int**, and hence generates code that passes only part of the pointer to **g**.

Pointer Variables And Constants As Arguments On Calls

Code that passes integer constants or variables as function arguments where pointers are expected may not work with Aztec C.

One common problem is attempting to pass the constant **int** 0 as a null pointer. For example, if **g** is again a function that is passed a pointer, the following code demonstrates one way to correctly pass a null pointer to **g**:

```
g((void *) 0);
```

(The code does not have to cast 0 to be a pointer to a **void**; casting it to be a pointer to any other type of object also would work.) If, instead, the code says:

```
g(0);
```

with the **-ps** option, a 16-bit null value is passed, which would be wrong.

Other Considerations

Internal Storage Of Numeric Data

Programs written for processors that store data items in least significant to most significant order may need to be changed. The MC68000 processor stores data in most significant to least significant order. This is true of both non floating point and floating point data.

The following short program illustrates a program that runs differently depending on the manner in which `int` data items are stored.

```
cput (c)
int c;
{
    write (1, &c, 1);
}
```

Problems can also occur reading data created in another environment where the data is stored in the reverse order.

The MC68000 requires that any memory access must be aligned on an even address unless it is a single byte access. The Aztec C compiler aligns nonbyte data items on even boundaries to avoid memory faults. Code that accesses nonbyte data through pointers that specify an odd memory address causes a system crash.

Converting Data

Programs reading data written in another C environment that does not force alignment will probably not produce correct results. Programs writing data that will be accessed in an environment that does not force alignment will likewise probably fail.

Some conversion will probably be needed to insert slack bytes to assure even alignment when importing data created for an unaligned environment, and to remove slack bytes when exporting data for an unaligned environment.

Most of the common 8-bit microprocessors store numeric data in an order that is the reverse of the MC68000. The 808x 16-bit processors and the PDP-11 also store numeric items this way.

Additional Features

Line Continuation

If the compiler finds a source line whose last character is the backslash character `\`, it considers the following line to be part of the current line, without the backslash. For example, the following statements define a character array containing the string **abcdef**:

```
char arr[]="ab\  
cd\  
ef";
```

String Concatenation

Character string literals that are adjacent tokens are concatenated into a single character string literal. For example, the following statements define a character array containing the string **abcdef**.

```
char arr[] = "ab" "cd" "ef";
```

In-line Assembly Code

Assembly language code can be interspersed within C source code by surrounding the assembly code with the statements **#asm** and **#endasm**.

For example,

```
main()  
{  
/* C code */  
#asm  
; assembly language code  
#endasm  
/* C code */  
}
```

Since **#asm** and **#endasm** are handled by the preprocessor, the compiler does not actually see the assembly code. Therefore, to assure that the correct code will be generated, it is a good idea to precede the **#asm** with a null statement (i.e., semicolon).

For a discussion of register usage by assembly language programs, see the Programmer's Information section of the **Assembler** chapter.

For a discussion of accessing macro definitions, arguments, and automatic and static variables from within a **#asm** block, see the **Technical Information** chapter.

Chapter 3 - Language Specification

Compatibility Issues

The Aztec C68K compiler for the Amiga is entirely compatible with the proposed draft ANSI standard for C. This means that porting programs which conform to the ANSI standard to the Aztec C68K compiler should be a very straightforward process, with few if any modifications to the code.

In addition to ANSI compatibility, Aztec C also has a number of library functions and compiler switches which provide a fair level of UNIX v7 C, UNIX System 3 C, UNIX System V C, and XENIX C compatibility.

Compatibility with Other Aztec Compilers

Within major release numbers, code may be easily moved from one implementation of Aztec C to another. For example, moving code from v3.6a of the Manx Amiga compiler to v3.6c of the Manx Macintosh compiler is fairly simple (except for the differences in machine-specific library functions). Where release numbers differ (i.e., 3.6c and 5.0a), code is generally upward compatible, but some changes may be needed to move code down to a lower release.

Note that compatibility between machines exists only with respect to source compilation and use of standard library functions. Machine-specific functions, such as Amiga Workbench calls, have no direct equivalents on other machines such as PCs or Macintoshes. The only library functions

which are portable across Aztec C implementations are those marked "Aztec" or "ANSI" in the library function chapter.

Keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while
pascal			

Operators

[] () . -> ++ -- & * + - ~ ! / % << >> < > <=
 >= == != ^ | && || ? : = *= /= %= += -= <<= >>=
 &= ^= |= , # ##

Pre-Processing Directives

#if	#ifdef	#ifndef	#elif
#else	#endif	#include	#define
#undef	#line	#error	#pragma
#	#asm	#endasm	

Type Information

The types supported by Aztec C68K and their sizes are listed in the following table.

Base Type Sizes

<i>TYPE</i>	<i>SIZE (in bits)</i>
char	8
short int	16
int	16 or (32)
long	32
float	32
double	(32) or 64
long double	32, 64, or (96)
function pointer	32
data pointer	32

In cases where there is more than one possible size, the default is delineated by (). Signed types are the same size as their unsigned counterparts.

Integers and Long Integers

Integer and long values are stored internally from most significant byte to least significant byte (i.e., the normal Motorola 68000 format). This can cause problems with programs ported from architectures which use other conventions for 16-and 32-bit storage, such as the 8086. For example, a program designed for the 8086 which accessed 16-bit values one byte at a time would not work properly. Programs that access **ints** and **longs** in the normal manner should function normally.

Integers and long integers are also always aligned on an even-address boundary. This alignment is required by the 68000 CPU when it has to access a 16 or 32-bit value. Programs which attempt to access byte-sized data in an integer manner will most likely crash the program. In this example

```
char *byte_nums = {0, 1, 2, 3, 4};  
int *word_nums, sum;
```

```
word_nums = (int *)byte_nums+1;  
sum = *word_nums + *(word_nums + 1);
```

the last line will cause a 68000 exception number 3 (address error).

Characters

The `char` type is always considered as signed by Aztec C. An unsigned character must always be explicitly declared as **unsigned char**. This may cause portability problems when porting code from other compilers/platforms. For example, the following code will not work as it would on a machine which considered `chars` unsigned:

```
char a = 255;

if (a > 30)
    printf ("a is bigger than 30\n");
else
    printf ("a is smaller than 30\n");
```

Although `a` appears to be set to 255, it is actually considered to contain -1 (255 translates to 11111111 binary, which is the two's-complement representation of -1). Therefore, the message **a is smaller than 30** will be printed. On a machine with unsigned characters, the message **a is bigger than 30** would be printed.

Structures

Due to alignment requirements of integer and long values, padding bytes, also known as "holes", may exist within a structure. For example:

```
struct a_struct {
    char a_char;
    int an_int;
};
```

In this structure, a padding byte will exist between the `a_char` and `an_int` elements to insure that `an_int` falls on an even address boundary.

Because of the existence of holes within structures, the `sizeof` operator will not always return what you would expect. In the above example,

```
sizeof (a_struct)
```

would return (assuming 32-bit ints) a 6, not a 5. This may cause some problems in porting code from environments such as the 8086 and 8-bit micros.

Bitfields

Bitfields are fully supported by Aztec C68K. Bitfields are numbered as per normal 68000 convention. For example:

```
struct {  
    unsigned int bf1 :1;  
    unsigned int bf2 :1;  
    unsigned int bf3 :4;  
} bf;
```

Here **bf1** will correspond to bit 0 in the first word in the structure **bf**, **bf2** will correspond to bit 1, and **bf3** will correspond to bits 2-5. As many bit fields as possible are put into a single word location. If a bit field entry has a length and position such that it would straddle two words, it will be moved entirely into the second word, and the remaining bits in the first word are considered to be undefined.

Enum Constants

By default, **enum** constants are always represented as an **int**. However, a compiler switch will cause **enums** to be sized to the smallest possible type (**char**, **short**, **int**, or **long**).

Const and Volatile

Type checking for the **const** keyword is fully supported by Aztec C68K. This means that the compiler will flag the following code as an error:

```
const char c_char = 10;  
c_char = 3;
```

The **volatile** keyword is also semantically as well as syntactically supported. The compiler will turn off all optimization for any variables of type **volatile**. More specifically, the compiler will never perform register optimizations for **volatile** variables. The value of the variable will always be fetched from the variable's location in main memory.

General ANSI-Compatible Features

Aztec C68k fully supports the ANSI standard. The following describes some of the more important ANSI features.

Function Prototyping and ANSI Function-Definition

ANSI function prototypes and ANSI-style function definitions allow the Aztec C compiler to identify incorrect specifications of function arguments. A function prototype is a function declaration which informs the compiler of the number and type of arguments that the function takes, and the function's return type. With this information, the compiler performs extensive type-checking on function arguments and identifies common errors, such as accidentally passing a character pointer instead of a character to a function like `putc()`.

There are three C constructs involved in function prototypes: function declarations, function definitions, and function calls. We will first consider the use of these components as described by Brain Kernighan and Dennis Ritchie in the book *The C Programming Language* (from here on referred to as "K&R"), which defines the version of C used by Aztec C prior to version 5.0:

FUNCTION DECLARATION - Function declarations in K&R only identify the return value of a function. This type of declaration will be referred to as an **OLD-STYLE** declaration. The following is an example

```
① double sin();  
②  
③ int positive_sin(d);  
④ double d;  
⑤ {  
⑥     d = sin (d);  
⑦     return (d > 0.0);  
⑧ }
```

Line 1 is a **DECLARATION** of the function `sin()`, and states that `sin()` returns a double-precision floating point number. Without this declaration, the compiler assumes that `sin()` returns an `int` instead of a `double`. This assumption would cause `d` to contain a random and

incorrect result. It is important to note that a function's return-type is the *only* information conveyed in old-style definitions.

FUNCTION DEFINITION - The function definition is the one and only place where a function is defined. The definition specifies the function's return type, arguments, and the actual code comprising the function (the body of the function). Using the above example, lines 3-8 comprise the full definition of the function `positive_sin()`. Lines 3 and 4 define the function name, return type, and arguments, and lines 5-8 comprise the function body.

In old-style function definitions, an important rule to remember is that arguments of type `short` or `char` are *always* converted to type `int`. Arguments of type `float` are *always* converted to `double`.

If you pass a function a variable of type `float`, for example, the compiler will automatically convert it to type `double` before passing it to the function (this is called **PROMOTION**). Thus, it's impossible to pass variables of types `char`, `short`, and `float` directly to a function when old-style definitions are used.

FUNCTION CALL - Line 6 in the above example shows a call to the function `sin()`, with the double-precision variable `d` as an argument, and `d` also receiving the return value of `sin()`. By looking at the function declaration of `sin()` on line 1, we know that `sin()` does indeed return a double value. If we tried to equate a variable of type `int` to `sin()`, the compiler would issue a warning message that the variable did not match the return type.

However, it is not apparent from the function declaration exactly how many arguments `sin()` takes, nor what their types may be. In fact, the compiler does not know the arguments received by `sin()` either, so *any* variable could be passed to `sin()` without any compiler warnings. Thus, we must rely on memory or reference manuals to insure that the correct arguments are passed to `sin()`.

ANSI Standard

The ANSI standard has extended the syntax of both the declaration and definition of functions to allow the number and types of arguments to be

specified in function declarations. For example, the above example could be re-coded as:

```
double sin(double x);

int positive_sin (d)
double d;
{
    d = sin (d);
    return (d > 0.0);
}
```

Notice that the first line now contains **double x** within the parenthesis. This informs the compiler that the **sin()** function takes an argument of type **double** (the **x** is there simply for readability; the compiler ignores it). The compiler will do appropriate error checking. Thus, if we tried to say:

```
d = sin (4);
```

the compiler will issue a warning message that the type of **4** (**int**) does not match the function's required argument type (**double**) and would then convert the **int** into a **double**.

Look at this a little closer, using another example:

```
void other (float size, long time, short stk);

void test (char c, float f, long l)
{
    other (c, 1, 1);
}
```

The first line indicates that **other** has no return value (**void**), and takes a **float**, **long**, and **short** argument (in that order). The declaration of function arguments has some important ramifications:

- Each argument will be converted, as if by assignment, to the corresponding type specified in the declaration. Thus, **c** will be converted from a **char** to a **float**, the integer constant **1** will be converted to a **long**, and **1** will be truncated from a **long** into a **short**.

- In addition, the compiler will warn that these three items are being coerced to fit the function declaration (these type of warnings can be turned off with compiler switches). The compiler will also check to insure that the correct number of arguments are being passed and will terminate with an error if too few or too many arguments are passed to the function.
- Default promotions are NOT done on **char**, **short**, and **float** values. Thus the **float** parameter in the above example will always be passed as a **float**, and not as a **double**. In essence, the prototype function declaration defines the default conversions to be done.

It is also important to note that when ANSI prototype-declarations are used, a new style of function definition is also used. Rather than declaring the names of the arguments in parenthesis, then declaring the argument types below that, the ANSI style declares the argument types and names together within the parenthesis (as is done on the third line). As a general rule, when function prototypes are used for a function, you should use the ANSI-definition style. In the absence of prototypes, you may use either style.

To summarize:

- If a function definition uses the new ANSI-style, arguments are treated exactly as specified in the argument list, regardless of whether a new or old-style declaration exists.
- If a function definition uses the old K&R style, arguments are treated as though they have been promoted.
- If there is an old-style or missing declaration, no checking on argument types is done.
- If an ANSI-prototype declaration is present with an old-style definition, then the declaration must use promoted variable types (**int** or **double** instead of **char**, **short**, or **float**).

Auto Aggregate Initialization

An AGGREGATE type is a generic term for a complex C type, and the term generally refers to structures and arrays. ANSI allows a feature known as AUTO-AGGREGATE INITIALIZATION, which means essentially that the aggregate types (structures and arrays) may be initialized as local (auto) variables. Two examples of auto-aggregate initialization would be:

```
struct st_tag {
    int a, b;
};

void funkier_func (void)
{
    struct st_tag a_struct = {0, 1};
    char name[] = "mike";
    .
    .
    .
}
```

The variables `a_struct` and `name` are the auto-aggregates that are initialized. Auto-aggregate initialization is not supported by K&R and is not implemented on many UNIX compilers.

Wide Strings and Literals

Wide literals and strings are used to address the problem of character sets which are too large to fit within the constraints of an 8-bit `char`. Wide character constants and string literals are specified by preceding the argument with an `l` or `L`, and are considered to be of type `wchar_t` (defined in `stddef.h`). The `wchar_t` type is defined by Aztec C68K to simply be of type `char`. For example:

```
wchar_t *cp = L"ABC";
```

is a declaration of wide string literal pointer. As is implied above, wide strings and literals are currently implemented exactly the same as normal characters and are included only to provide full ANSI compatibility.

Structure Assignment and Structure Passing

Assignment of one structure to another is supported. In the example:

```
struct st {
    int st_type;
    char *st_name;
} st_first = {5, "Hey!"};

struct st st_second;
void func (void)
{
    st_second = st_first;
}
```

the structure `st_second` will take on the value 5 for `st_second.st_type`, and `Hey!` for `st_second.st_name`.

Passing of full structures to a function is also legal; the compiler simply copies the contents of the structure onto the stack. A function can also return a structure as its value.

Trigraphs

TRIGRAPHS are three-character alternate representations of certain non-alphanumeric characters. They exist primarily for systems which cannot input some of the less common non-alphanumeric characters directly from the keyboard. The supported trigraphs are:

<i>Trigraph</i>	<i>Character it represents</i>
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	
??>	}
??-	~

By default, Aztec C68K does not parse trigraphs. However, tri-graph parsing may be turned on by means of a compiler switch.

Escape Sequences

The following character escape sequences are supported by Aztec C68K:

<code>\'</code>	single-quote.
<code>\"</code>	double-quote.
<code>\?</code>	question-mark.
<code>\\</code>	backslash.
<code>\a</code>	Alert - flashes the Amiga screen briefly.
<code>\b</code>	Backspace - Moves the cursor back one space.
<code>\f</code>	Form feed - Move to the next physical page.
<code>\n</code>	New line - Move to the start of the next line.
<code>\r</code>	Carriage return - Move to the start of the current line.
<code>\t</code>	Horizontal tab - Move to the next tab position.
<code>\v</code>	Vertical tab - Move to the same horizontal position on the next line.
<code>\x</code>	Hex integer - the digits following the <code>\x</code> define a hexadecimal constant.
<code>\digits</code>	Octal integer - the digits following the <code>\</code> define an octal constant.

Long Character Constants

Aztec C68K recognizes long character constants. The following code will put the ASCII values for **a**, **b**, **c**, and **d** into the long value **l**:

```
long l;
```

```
l = 'abcd' ;
```

Note that this is different than a string, in that each letter is considered to be a character-sized constant, and there is no null appended to the end.

Register Variables

The Aztec C compiler supports a total of nine register variables. Specifically, six data registers (d2-d7, for general non-pointer data) and three address registers (a2, a3, a6, for pointers) are reserved for register variables.

The compiler can automatically allocate register variables based on the relative usage of each variable, without an explicit **register** declaration. This behavior is entirely optional and is turned off by default.

The following types may be declared as register variables:

- **char, short, int, long, pointer, float, double**

In addition to these, the unsigned counterparts of the above may also be declared as **register**.

Sizeof and Size_t

The ANSI standard defines a type known as **size_t**, which is used by a number of library functions, and is also the return type of the **sizeof** operator. **size_t** is always an integral type, and should be large enough to contain the size of the largest data element addressable by the compiler. On the Aztec C68K system, **size_t** is defined to be a **long**. Because **size_t** returns a **long**, data objects in the large data model may be of any size (in the small data model, total data is limited to 64K).

Symbol Names

All symbol names, both of internal and external linkage, have 31 significant characters. Symbols with global scope have a "_" prepended to their names internally. This "_" is invisible at the C source level, but is necessary when accessing C variables from assembly language.

Preprocessor Features

Predefined Symbols

The pre-defined preprocessor symbols **__FILE__**, **__LINE__**, and **__FUNC__** are fully supported, as per ANSI.

- `__FILE__` is a string which expands to the name of the current file being compiled
- `__LINE__` expands to an integer representing the current line number
- `__FUNC__` expands to a string which specifies the current function being compiled.

For example, consider the following C file called `test.c`:

```
void
funky_func (void)
{
    printf ("File = '%s'", __FILE__);
    printf ("Line = %d", __LINE__);
    printf ("Function = '%s'", __FUNC__);
}
```

The function `funky_func` will produce this output when it is called:

```
File = 'test.c'
Line = 5
Function = 'funky_func'
```

There are several symbols specific to Aztec C68K that are always predefined by the compiler. These are: `MPU68000`, `MCH_AMIGA`, and `AZTEC_C`. Also, a symbol named `VERSION` is defined and set to 500 (version 5.00).

Other symbols (also specific to Aztec C) that may be defined are:

<i>SYMBOL</i>	<i>Defined when...</i>
<code>_INT32</code>	Integers are 32-bits in length
<code>_LARGE_CODE</code>	The large code model is being used.
<code>_LARGE_DATA</code>	The large data model is being used.

String Concatenation

String-concatenation is a pre-processor feature which allows two or more strings which occur next to each other in C source to be concatenated together and treated as one string. For example:

```
#define ERROR "ERROR: "  
    printf (ERROR "Hey you! "  
        "A system error has occurred!!");
```

In this example, the `printf` call would print out:

```
ERROR: Hey you!  A system error has occurred!!
```

There are two useful applications for string concatenation:

- Long strings may be separated into several short strings. This is often desirable to keep the indentation consistent within the body of a function. This is why **Hey you!** and **A system error has occurred!!** are split—to preserve the indentation of the surrounding program, thereby making it easier to read.
- `#defines` may be used for common prefixes for strings. In the above example, the string **ERROR:** is assigned to a macro named **ERROR**, which might be a standard prefix for error messages.

Line Continuation

Logical lines that cannot fit on one line may be continued by use of the line continuation character, `\` (a backslash), being placed at the end of the physical line. For example, the following statements define a character array with the contents **abcdef**:

```
char an_array[] = "ab\  
cd\  
ef";
```

It is useful to note that, in most cases, string concatenation is a preferable method to line continuation. This is because string concatenation can preserve a program's indentation, whereas line continuation cannot. The tabs or spaces used for indenting would show up in the string.

Chapter 4 - Assembler

The Manx Aztec C Assembler, `as`, translates assembly language source statements into relocatable object code.

Assembler source statements are read from an input text file and the object code is written to an output file. A listing file is written if requested. The relocatable object code must be linked by the Manx Aztec C Linker, `ln`, before it can be executed. At linkage time it may be combined with other object files and run time library routines from system or private libraries. Object modules produced from C source text and assembler source text can be combined at linkage time into a composite module.

Assembly language routines are generally not required when programming in C. Assembly language routines should only be necessary where critical execution time or critical size requirements exist. Some system interfacing or low level routines may also require assembler code. Even when assembler use is indicated, it may not be necessary to write separate assembler files. The `cc` command supports in-line assembler code.

Information on the MC68000 and MC68020 architecture and instructions can be found in the *Motorola MC68000 16/32-bit Microprocessor Programmer's Reference Manual* and the *Motorola MC68020 32-bit Microprocessor User's Manual*, respectively (Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632).

Operating Instructions

The assembler is started by entering the command line

as [*-options*] *filename*

where [*-options*] specify optional parameters and *filename* is the name of the file to be assembled.

The assembler is also invoked by the C Compiler to assemble its output file.

The assembler reads assembly source statements from the input file, writes the translated relocatable object code to an output file, and if requested writes a listing to an output file. The assembler also merges assembly code from other files upon encountering an **include** directive.

EXECUTION ENVIRONMENT

Processor Support

The assembler supports the MC68010, MC68020, MC68030, and the MC68881, MC68851, instruction sets and addressing modes, in addition to those of the MC68000.

The assembler defaults to assuming that only the MC68000 instructions are valid. The **machine**, **mc68881**, and **mc68851** directives enable and/or disable the additional instructions and addressing modes.

INPUT FILE

The input file is a text file that is usually created by a text editor or the Manx Aztec C Compiler. The input file resides in the current directory. If it does not, a fully qualified or partially qualified path name can be prefixed to the filename to designate the source directory. Although .asm is the recommended filename extension, any extension is acceptable.

The specification:

as x

assembles the file **x.asm** if it exists. If the file **x.asm** does not exist then the file **x** is assembled. If the file **x** does not exist or if file **x** cannot be assembled, then error messages will be returned.

OBJECT CODE FILE

The assembler writes the object code it produces to a file. By default this file is placed in the directory that contains the source file, and its name is derived from that of the input file by changing the extension to **.o**.

To write the object code to a file in another directory, and/or to a file having another name, use option **-o**.

For example, the following command assembles the source in **prog.asm**, sending the object code to the file **new.obj**. The latter file is placed in the current directory, because option **-o** does not specify otherwise.

```
as -o new.obj prog.asm
```

LISTING FILE

If you specify option **-l**, the assembler produces a listing file with the same root as the input file and a filename extension of **.lst**. The listing file displays the source statements and their machine language equivalent. The listing also indicates the relative displacement of each machine instruction.

OPTIMIZATIONS

By default, the assembler performs some optimizations on an assembly language source file, by making two passes through the assembly source file.

Option **-n** disables some optimization, thereby allowing the assembler to run faster because it makes only a single pass through the source and because it does not optimize the code. However, usage of the **-n** option makes the resultant code larger and slower.

Optimizations affect the following instructions:

branches converts long branches to short if possible and deletes branches to the following location

movem	If there are no registers, deletes the instruction. If there is only one register, substitutes the shorter move instruction. Note that the move instruction alters condition codes, while movem does not.
jsr	substitutes bsr , if possible.
jmp	substitutes bra if possible.

SEARCHING FOR INCLUDE FILES

By default the assembler searches only the current directory for files specified in **include** statements. It also searches a user-specified sequence of directories for such files, thus allowing program source files and header files to be contained in different directories.

Option **-i** and the **INCLUDE** environment variable define the directories the assembler searches for **include** files.

The assembler automatically searches only the current directory for an include file if the following conditions are met:

- (1) the assembler is started without option **-i**
- (2) **INCLUDE** is not an environment variable, and
- (3) the **include** statement does not specify the drive and/or directory containing the file.

If an **include** statement specifies either the drive or directory, only that location is searched for the file.

Option -i

Option **-i** defines a single directory to be searched for a file specified in an **include** statement. The path descriptor follows **-i** with no intervening blanks. For example, the specification

```
as -isys:db/include prog1
```

directs the assembler to search the **sys:db/include** area when looking for an include file. If desired, more than one **-i** may be used, thus defining multiple directories to be searched.

INCLUDE Environment Variable

The **INCLUDE** environment variable, if it exists, also defines directories to be searched for include files.

The **INCLUDE** variable consists of the names of the directories to be searched, separated by semicolons. For example, the following command creates the **INCLUDE** environment variable, defining three directories to be searched:

```
set INCLUDE=work:include;work:;sys:include
```

These directories are (1) the **include** directory on the **work:** volume (2) the **root** directory on the **work:** volume, and (3) the **include** directory on the **sys:** volume.

Include Search Order

When the assembler encounters an **include** statement, it searches directories for the file specified in the statement in the following order:

- (1) current directory
- (2) directories specified in option **-i** in the order listed on the line that started the assembler
- (3) directories specified in the **INCLUDE** environment variable in the order listed

MODULE MEMORY MODEL

Options **-c** and **-d** instruct the assembler to use the large code and large data models, respectively. For a discussion of these models, see the **Compiler** chapter of this manual.

Assembler Options

SUMMARY OF OPTIONS

-a	Forces total size of code and data to be aligned to a 4-byte boundary instead of the default 2-byte boundary.
-c	Makes large code the default code memory model. May be overridden by the near code and/or far code directives.
-d	Makes large data the default data memory model. May be overridden by the near data and/or far data directives
-ename[=val]	Creates an entry in the symbol table for <i>name</i> and assigns it the constant value <i>val</i> . If <i>val</i> is not specified, <i>name</i> is assigned the value 1.
-iarea	Defines an area to be searched for files specified in an include statement.
-l	Generates an assembly listing file.
-m	Metacomco compatibility option.
-n	Does not optimize object code.
-o filename	Specifies name of output object module.
-v	Verbose option. Generates memory usage statistics.
-ZAP	Used primarily by the Aztec C compiler to direct the assembler to delete the input file after processing.

DESCRIPTION OF OPTIONS

Option -c

If you use this option, the assembled module uses the large code memory model.

Option -d

If you use this option, the assembled module uses the large data memory model.

Option -o *filename*

If you use this option, the assembler sends the object code to *filename*. If option -o is not specified, the assembler sends the object code to a file whose name is derived from that of the assembler source file (by changing the extension to .o). In this case, the file is placed in the directory containing the source file.

Option -i

The -i option specifies where files referred to by the include directive are kept.

If you use option -i, the assembler searches a specified area for files included in the source code.

The name of the area should immediately follow the -i with no intervening spaces. For example, the following defines the directory **source/inc** on volume **sys:** as an area to search for include files:

```
-isys:source/inc
```

For more details, see page 4-4.

Option -l

If you specify option -l, the assembler generates a listing file. The name of the file to which the listing is sent is derived from that of the source file by changing the extension to .lst. The listing file is placed in the directory containing the source file.

Source Program Structure

There are four types of Assembler statements:

- (1) Comments
- (2) Executable Instructions
- (3) Directives
- (4) Macro Calls

COMMENTS

A comment can appear after a semicolon or after the operand field. For example:

```
; this is a comment  
link a6, #.2 ;this is also a comment
```

A line used for a comment may alternatively begin with an asterisk.

EXECUTABLE INSTRUCTIONS

Executable instructions have the general format:

label operation operand

LABELS

Assembler labels can be any length. External labels are only significant for the first 31 characters. Any additional characters are ignored. Valid label characters include letters, numbers, or the special characters ". "and" _". A label cannot begin with a digit unless it is a temporary label. Labels that do not start in the first column require a colon suffix. **Note:** C prefixes all external symbols with an underscore. For example, a C data object, `int i`, will appear as `_i` at the assembly level and should be referenced as such.

OPERATIONS

The assembler recognizes all of the mnemonics found in Motorola's reference manuals for the 68000,68020 processors and the 68881,68851 coprocessors

To specify a length for instructions that support multiple lengths, suffix the instruction mnemonic with:

- .B** 8-bit operands
- .W** 16-bit operands
- .L** 32-bit operands

OPERANDS

The operand field consists of one expression, or two expressions separated by a comma with no embedded spaces. An expression comprises register mnemonics, symbols, constants, or arithmetic combinations of symbols or constants. In addition certain 68020 instructions require special operand syntax.

Symbols or labels represent relocatable or absolute values. An absolute value is one whose value is known at assembly time. A relocatable value is one whose value is not known until the program is actually loaded into memory for execution.

Relocatable expressions can only be expressed arithmetically as sums or differences. The difference between two relocatable expressions is absolute. The result of summing two relocatable expressions is undefined.

There are five types of constants: octal, binary, decimal, hexadecimal and string. An octal constant is expressed as an @ followed by a string of digits from the set 0 through 7 such as:

@123 or @777.

A binary constant is expressed as a % followed by a string of ones and zeroes, such as

%10101 or %11001100

A decimal constant is a string of numbers. A hexadecimal constant is a \$ followed by a string of characters made up of numbers or letters from a through f such as

\$ffff or \$1a2e

A string constant is any string of characters enclosed in single quotes such as

`' abdc '`

Register mnemonics include the data registers **d0** through **d7**; the address registers **a0** through **a7**; **sp** (same as **a7**) the stack pointer; **pc** the program counter (forces PC relative mode); **sr** the status register; **ccr** the condition code register ; and the user stack pointer **usp**.

The assembler supports addition (+), subtraction (-), multiplication (*), division (/), shift right (>), shift left (<), unary minus, and (&), or (|). It also supports inclusive or (!), exclusive or (^), bitwise not (~), and modulo (/ /).

The order of precedence is innermost parenthesis, unary minus, shift, and/or, multiplication/division, addition/subtraction, bitwise, and logicals.

DIRECTIVES

The following paragraphs describe the directives that are supported by the assembler.

blanks

blanks {on | off}
blanks {yes | no}
blanks {y | n}

This directive controls whether the assembler accepts blanks or tabs in the operand field of the instruction.

The default setting of **off** treats a blank as the end of the operand field.

The **blanks on** setting allows blanks to be placed between any two complete items. With this setting all comments must be preceded by a ;.

clist and noclist

These directives specify whether or not statements should be included in the listing file, when the statements were not assembled as a result of

assembler conditional statements. By default, such statements are not listed.

cnop

label cnop n1,n2

This directive is used to force alignment on any boundary at a particular offset.

The first value, *n1*, is an offset while the second value, *n2*, specifies the alignment to be used as the base of the offset. For example, to align to an even word boundary:

cnop 0,2

while to align to a long word boundary:

cnop 0,4

and finally to align to a word beyond a long word boundary:

cnop 2,4

Note that this will only take effect relative to the beginning of the current module's code or data. Normally, the linker will not align individual modules to long word boundaries. So, for this directive to work, it must be the first module linked into the program, or the linker's **-a** option must be used.

cseg

The assembly code following this directive is output into the code segment of the program output file.

dseg

The assembly code following this directive is placed in the initialized data segment of the program file.

dc - Define Constant

<i>[label]</i>	dc.b	<i>value [value, value ...]</i>
<i>[label]</i>	dc	<i>value[,value, value ...]</i>
<i>[label]</i>	dc.w	<i>value[,value, value ...]</i>
<i>[label]</i>	dc.l	<i>value[,value ,value ...]</i>
<i>[label]</i>	dc.b	<i>"string"</i>

The **dc** directive causes one or more fields of memory to be allocated and initialized. Each *value* operand causes one field to be allocated and then to be initialized with the specified value. A *value* can be an expression. An expression may contain forward references.

For command programs, a value can contain a reference to a memory location whose address will not be known until the program is loaded into memory. In this case, an item for this value will be added to the program's relocation table. When the program is loaded, the field containing this value will be set to the correct value.

Each field for a particular **dc** directive is the same length. A period followed by **b**, **w**, or **l** can be appended to a directive, defining the field length to be one, two, or four bytes, respectively.

If the field length is not specified in this way, it defaults to 2 bytes.

Fields that are two or four bytes long are aligned on word boundaries.

The last form listed for **dc** allocates a field having exactly the number of characters in the string, and places the string in it. **Note:** Trailing null characters must be explicitly requested, for example

```
dc.b "Hello",0.
```

dcb - Define Constant Block

<i>[label]</i>	dcb.b	<i>size[,value]</i>
<i>[label]</i>	dcb	<i>size[,value]</i>
<i>[label]</i>	dcb.w	<i>size[,value]</i>
<i>[label]</i>	dcb.l	<i>size[,value]</i>

The **dcb** directive allocates a block of storage containing *size* fields, and initializes each field with *value*. If *value* is not specified, it is assumed to be 0.

Each directive is the same length. A period followed by **b**, **w**, or **l** can be appended to a directive, defining the field length to be one, two, or four bytes, respectively. If the field length is not specified in this way, it defaults to 2 bytes(**.w**).

Fields that are two or four bytes long are aligned on word boundaries.

ds - Define Storage

<i>[label]</i>	ds.b	<i>size</i>
<i>[label]</i>	ds	<i>size</i>
<i>[label]</i>	ds.w	<i>size</i>
<i>[label]</i>	ds.l	<i>size</i>

This directive allocates a block of storage containing *size* fields, and sets each field to 0.

Each field for a particular **ds** directive is the same length. A period followed by **b**, **w**, or **l** can be appended to a directive, defining the field length to be one, two, or four bytes, respectively.

If the field length is not specified in this way, it defaults to 2 bytes(**.w**).

entry

[label] **entry** *symbol*

This directive defines the entry point of the program. Only one entry can be declared per program. If no entry point is defined, the first instruction of the first module becomes the default entry point. The entry point must be in the first 32K of the root segment.

end

This directive defines the end of the source statements.

equ

label **equ** *expression*

This directive assigns the value of the expression on the right to the label on the left.

equr

[label] equr register

This directive allows a register to be referenced by an alternate name. Reference to the new name is made without regard to case.

even

[label] even

This directive forces alignment to a word (16 bit) boundary.

fail

fail

This directive causes the assembler to generate an error for this line. This can be used in macros which detect an incorrect number of arguments and wish to prevent assembly.

far code

This directive causes the assembler to generate code for the large code memory model. Absolute addressing will be used with fixups being done by the startup code.

far data

This directive causes the assembler to generate code for the large data memory model. Absolute addressing will be used with fixups being done by the startup code.

freg

[label] freg register list

This directive is like the **reg** directive, except that it is used to specify the floating point registers of the MC68881. The list is either composed of the floating point registers **fp0** through **fp7** or of the floating point control registers **fpcr**, **fpsr**, and **fpiar**, but not both.

ifc and ifnc

```
ifc  'string1','string2'  
ifnc 'string1','string2'
```

These conditionals check to see if the two strings are equal. If they are, the **ifc** will assemble the following code, while **ifnc** will skip it.

ifd and ifnd

```
ifd      symbol  
ifnd     symbol
```

These conditionals check to see if the specified symbol has been defined or not. If the symbol has been defined, then **ifd** will assemble the following code, while **ifnd** will not.

if, else, and endc

```
if  test  
...  
[else]  
...  
endc
```

These directives are used to allow conditional assembly of parts of the input file. The general form of the if test is:

```
exp  
exp == exp || exp = exp  
exp != exp || exp <> exp  
'str1' == 'str2' || 'str1' = 'str2'  
'str1' != 'str2' || 'str1' 'str2'
```

If the test result is true, then the lines up to an **else** or **endc** are assembled. If there is an **else**, then lines up to the **endc** are skipped. The skipped lines are not displayed in the listing file unless the **clst** directive has been used. If the test is false, then lines are skipped until an **else** or **endc** is encountered. If it is an **else** then the following lines up to an **endc** are assembled.

near code

This directive causes the assembler to generate code for the small code memory model. It is the default unless the **-c** option, or **far code** directive, is used.

near data

This directive causes the assembler to generate code for the small data memory model. It is the default unless the **-d** option, or **far data** directive, is used.

other ifs

```
ifeq absolute_expression
ifgt absolute_expression
ifge absolute_expression
ifle absolute_expression
iflt absolute_expression
ifne absolute_expression
```

These conditionals perform a comparison of the value of the absolute expression to zero. If the specified condition is true, then the following assembly language is processed, otherwise it is skipped.

include

```
include 'filename'
include "filename"
include <filename>
```

This directive causes the contents of the file specified to be processed by the assembler as if they had appeared at the same relative location as the include statement.

global and bss

```
[label]    global symbol, size
[label]    bss    symbol, size
```

These directives reserve storage for uninitialized data items. The area is reserved in the uninitialized data area. If **global** is used then the data item

is known to other modules that are external to the routine. If **bss** is used then the data item is local to the routine in which it is defined.

If a **global** is defined in more than one module, the linker will generate the error message:

each external name must have exactly one definition..

A symbol that appears in both a **global** and a **public** directive is located in the initialized data area and the global statements size parameters are ignored.

list and nolist

The directives **list** and **nolist** turn on and off, respectively, the listing of assembly language statements to the listing file.

mlist and nomlist

The directives **mlist** and **nomlist** specify if the assembly language statements generated by a macro expansion should be written to the listing file.

machine

```
machine mc68000
machine mc68010
machine mc68020
```

This directive enables or disables the additional instructions and addressing modes associated with different processors in the MC68000 family.

macro and endm

```
macro symbol
...
text
...
endm
```

or

symbol macro

```
...  
text  
...  
endm
```

symbol is entered in the assembler opcodes table. The text between the **macro** and **endm** is saved in memory. When *symbol* is encountered as an opcode the text is placed in line.

Up to nine arguments can be specified in one of two ways. There is also an argument **0** which refers to the extension on the **macro** directive when it was invoked. The arguments are referenced in the macro text as either %0 through %9 or \0 through \9. In expanding a macro symbolic argument references are replaced by their actual value.

The assembler also has facilities for generating unique labels within a macro. When the \@ is specified within a macro, the assembler will generate labels of the form .*nnn* where *nnn* will have a unique value for each invocation of the macro.

The symbol **narg** is a special assembler symbol that indicates the number of arguments specified when the macro is invoked. Outside of macro definitions, the value of **narg** is 0.

Macro arguments that contain a space or comma can be enclosed in bracketing "<" and ">" characters.

mc68851

This directive enables the **mc68851** memory management instructions to be recognized and assembled by the assembler.

mc68881

This directive enables the **mc68881** floating point instructions to be recognized and assembled by the assembler.

mexit

Upon encountering this directive, expansion of the current macro stops and the assembler scans for the statement following the **endm** directive.

public

[label] public symbol[,symbol..]

This directive identifies the specified symbols as having external scope. These symbols are visible to the linker and are used to resolve references between modules. The type of the symbol is CODE if it appears within the code segment and DATA if it does not.

reg

label reg register list

This directive assigns the value of the register list to the label. Forward references are not allowed. A register list consists of a list of register names separated by the / character. The - character may be used to identify an inclusive set of registers. This directive is generally used with the **movem** instruction.

The following are valid register lists:

***a0-a3/d0-d2/d4
a1/a2/a4/a6/a0-a2***

section

***[label] section name, CODE
[label] section name, DATA***

This directive performs the same functions as the **cseg** and **dseg** directives. The name parameter, if present, is ignored at the current time. The type parameter is used to switch from **CODE** and back again. If only a name parameter is specified, the type defaults to **CODE**.

set

label set expression

This directive assigns the value of the absolute *expression* to the symbol specified by *label*. This definition is similar to the **equ** directive, with the exception that this symbol's value can be changed with another **set** direc-

tive. This includes expressions referencing the current value of the symbol itself. For example:

```
sym      set      sym+1
```

takes the current value of **sym** and adds 1 to it, which then becomes the *new* value of **sym**.

ttl

```
ttl title_string
```

This directive sets the title of the current module being assembled. This directive is implemented for compatibility with other assemblers and has no effect at the current time.

xdef and xref

```
xdef      symbol  
xref      symbol
```

These directives are used to specify the definition and reference of global symbols.

MACRO CALLS

Macro calls consist of a macro name with an optional label, extension, and arguments, in this form:

```
[label]   macroname   [.ext]  [arg1, arg2...]
```

The optional extension consists of a "." followed by any valid 680X0 extension, such as **w**, **l**, etc.

Up to nine arguments may be passed to the macro.

See the **macro and endm** section of this chapter for more information regarding macros.

Interfacing With C

Interfacing 68000 assembly language routines with C is relatively easy. The linkage conventions are straightforward and simple.

REGISTER CONVENTIONS

It is the responsibility of an assembly language subroutine to preserve the values in registers **a2** through **a5** and **a7**, as well as registers **d4** through **d7**.

Register **d0** through **d3** and **a0**, **a1** and **a6** are available as work registers. There is no need to preserve the values of work registers for other routines.

Register **d0** contains the return value of the subroutine if the return type is non floating point. If the return value is floating point, then the return value is in the register pair **d0-d1**.

ARGUMENTS TO SUBROUTINES

Upon entry register **a7 (sp)** points to the stack. The first item on the stack is a 32 bit absolute return address. The second item on the stack is the first argument to the subroutine, followed by the second, and so on. Arguments to C subroutines are passed by value. Therefore character, integer, long, and floating point arguments are copied onto the stack by value. If the functions are not prototyped, character items are promoted to type **int** before being pushed on the stack, otherwise character items are passed normally.

Variable Names

The compiler attaches a leading underscore to all variable names. Therefore, in order to access these labels or names from within an assembly file, the programmer must prefix the labels with an underscore. The labels must also be declared as **public** or followed by a **#** which makes them **public**.

Returning from a C Function

To return from a C function, it is necessary to do the following:

- restore the values of registers **d4** through **d7** as well as registers **a2** through **a5** and **a7** to the values they had at entry to the routine;
- to place the return value, if any, in register **d0** or in **d0 + d1** if floating point; and
- to execute an **rts** instruction.

It is not necessary to restore values to registers that were not changed.

Upon return to the calling routine, the stack still contains the arguments that were passed to the subroutine. The first argument is on the top of the stack.

Chapter 5 - Linker

The Manx linker creates executable programs by linking together the pieces of the program that are compiled and assembled and are in relocatable object format.

The linker has two functions:

- It ties together the pieces of a program that were compiled and assembled separately
- It converts the linked pieces to a format that can be loaded and executed.

The Manx Assembler must first create the pieces.

From those pieces, the linker can create several types of executable objects:

- **CLI Programs**
- **Workbench Programs**
- **Drivers**, which other programs call to access devices

Introduction to Linking

For those with limited exposure to linkers, this section introduces linking in general and the Manx linker in particular. If you are experienced or wish to skip this section, go right to “Using the Linker”.

LINKING Hello.o

It is very unusual for a C program to consist of a single, self-contained module. Consider a simple program that prints

```
hello, world
```

using the function, `printf()`. The terminology here is precise: `printf()` is a function and not an intrinsic feature of the language. It is a function that you might have written but did not have to, because Aztec C already provides it in the file, `c.lib`. This file is a library of all the standard I/O functions. It also contains many support routines that are called in the code generated by the compiler. These routines aid in integer arithmetic, operating system support, etc.

When the linker sees that a call to `printf()` is made, it pulls the function from the library and combines it with the `hello, world` program. The link command looks like this:

```
ln hello.o c.lib
```

or

```
ln hello.o -lc
```

When `hello.c` is compiled, calls are made to some invisible support functions in the library. So linking without the standard library causes some unfamiliar symbols to be undefined.

Almost all programs need to be linked with some variant of `c.lib`.

THE LINKING PROCESS

Since the standard library contains only a limited number of general purpose functions, all but the most trivial programs are certain to call

user-defined functions. It is up to the linker to connect a function call with the definition of the function somewhere in the code.

In the example given below, the linker finds two function calls in **file1**. The reference to **func1** is "resolved" when the definition of **func1** is found in the same file. However, the reference to **func2** is unresolved because **func2** is found in another file, e.g., **file2.o**. Therefore, the following command

```
ln file1.o c.lib
```

causes an error indicating that **func2** is an undefined symbol. To be successful, the linkage must include **file2.o** as follows:

```
ln file1.o file2.o c.lib
```

Example

File 1

```
main ()
{
    func1 ();
    func2 ();
}

func1 ()
{
    return;
}
```

File 2

```
func2 ()
{
    return;
}
```

Libraries

A library is a collection of object files put together by a librarian. Libraries intended for use with the linker must be built with the Manx librarian, **lb**. This utility is described in the **Utilities** chapter.

The linker also recognizes AmigaDos libraries (**Amiga.lib**) See the discussion of these in the appropriate chapter of your *Aztec C Library Manual*.

All the object files specified to the linker are pulled into the linkage; they are automatically included in the final executable file. However, when a library is encountered, it is searched. Only those modules in the library that satisfy a previous function call are pulled in.

The **CLIB** environment variable, used to specify where libraries may be found, supports multiple entries when they are separated by an exclamation point (!). For example, if libraries are in both the **sys2:lib** and **ram:** directories, then **CLIB** would be defined like this:

```
set CLIB=sys2:lib!ram!
```

The linker automatically adds an **.o** extension to files that have no extension. It checks all directories defined in the **CLIB** environment variable and then the current directory .

EXAMPLE

Consider the "hello, world" example. Having looked at the module, **hello.o**, the linker has built a list of undefined symbols.

This list includes all the global symbols referenced but not defined. Global variables and all function names are considered to be global symbols.

The list of undefineds for **hello.o** includes the symbol **printf()**. When the linker reaches the standard library, this is one of the symbols it will be looking for. It discovers that **printf()** is defined in a library module whose name also happens to be **printf()**. (There is no relation between the name of a library module and the functions defined within it.)

The linker pulls in the **printf()** module in order to resolve the reference to the **printf()** function.

Files are examined in the order in which they are specified on the command line. So the following linkages are equivalent:

```
ln hello.o  
ln c.lib hello.o
```

Since no symbols are undefined when the linker searches **c.lib** in the second line, no modules are pulled in. It is good practice to leave all libraries at the end of the command line, with the standard library last of all.

The Order of Library Modules

For the same reason, the order of the modules within a library is significant. The linker searches a library once, from beginning to end. If a module is pulled in at any point, and that module introduces a new undefined symbol, then that symbol is added to the running list of undefineds. The linker does not search the library twice to resolve any references that remain unresolved. A common error lies in the following situation:

Module of Program	References (function calls)
<code>main.o</code>	<code>getinput</code> , <code>do_calc</code>
<code>input.o</code>	<code>gets</code>
<code>calc.o</code>	<code>put_value</code>
<code>output.o</code>	<code>printf</code>

Suppose we build a library to hold the last three modules of this program. Then our link step looks like this:

```
ln main.o proglib.lib c.lib
```

But it is important that `proglib.lib` is built in the right order. Assume that `main()` calls two functions, `getinput()` and `do_calc()`. `getinput()` is defined in the module `input.o`. It in turn calls the standard library function, `gets()`. `do_calc()` is in `calc.o` and calls `put_value()`. `put_value()` is in `output.o` and calls `printf()`.

The following paragraphs describe what happens at link time if `proglib.lib` is built in the order shown below:

```
proglib.lib:  input.o
              output.o
              calc.o
```

After `main.o`, the linker has `getinput` and `do_calc` undefined (as well as some other obscure functions in `c.lib`). Then it begins the search of `proglib.lib`. It looks at the library module, `input`, first. Since that

module defines `getinput`, that symbol is taken off the list of undefineds. But `gets` is added to it.

The symbols `do_calc` and `gets` are undefined when the linker examines the module `output`. Since neither of these symbols is defined there, that module is ignored. In the next module, `calc`, the reference to `do_calc` is resolved but `put_value` is a new undefined symbol.

The linker still has `gets` and `put_value` undefined. It then moves on to `c.lib`, where `gets` is resolved. But the call to `put_value` is never satisfied. The error from the linker looks like this:

```
Undefined symbol: _put_value
```

This means that the module defining `put_value` is not pulled into the linkage. The reason, as we saw, was that `put_value` was not an undefined symbol when the `output` module was searched. This problem would not occur with the library built this way:

```
proglib.lib:  input.o
              calc.
              ooutput.o
```

The `ord` command (described in the **Utilities** chapter) calculates a workable order for a collection of `.o` files.

Occasionally it becomes difficult or impossible to build a library so that all references are resolved. In the example, the problem could be solved with the following command:

```
ln main.o proglib.lib proglib.lib c.lib
```

The reason this is not the most satisfactory solution is that the linker must search the library twice, thus lengthening linking time.

The most common cause of cycles is illustrated by the following example:

<code>f.c:</code>	<code>g.c:</code>
<code>int a;</code>	<code>extern int a;</code>
<code>f()</code>	<code>g()</code>
<code>{</code>	<code>{</code>
<code>g();</code>	<code>a = 4;</code>
<code>}</code>	<code>}</code>

Here, `f()` invokes `g()` so file `g.o` should follow `f.o` in the library. However, `g()` references `a`, so `f.o` must follow `g.o`.

The solution to this problem is quite straightforward, however:

<code>f.c</code>	<code>g.c:</code>	<code>a.c:</code>
<code>extern int a;</code>	<code>extern int a;</code>	<code>int a;</code>
<code>f()</code>	<code>g()</code>	
<code>{...g();...}</code>	<code>{...a = 4; }</code>	

By moving the definition of `a` to a separate file, the order `f.o`, `g.o`, `a.o` allows all references to be resolved.

When there is no way to remove a cycle, the `ord` utility announces that fact and chooses `a.o` to break the cycle. The output of `ord` then contains duplicates.

You can then try using the splitting technique above, or use the output from `ord` with `lb` to construct a library with duplicates of some modules.

Using the Linker

As mentioned in the introduction to this chapter, the linker can create several types of executable programs. Much of the actual use of the linker is the same, regardless of the type of program generated.

STARTING THE LINKER

Start the linker by entering:

```
ln [-options] file1 file2 ...
```

where `[-options]` are linker options and `file1 file2 ...` are the names of files containing object modules and libraries of object modules.

For example, the most basic linker command which creates an executable "hello, world" program by linking object code in `hello.o` with modules in `c.lib` is

```
ln hello.o c.lib
```

This creates a command program in the file `hello`.

INPUT FILES

The linker scans the input files in the order in which they are specified on the command line. If a file contains a single object module, the module is automatically included in the program being built.

If an input file is a library of object modules, the linker makes one pass through the library, looking for modules containing a function or global variable that has been called by an already-included module and that is not in any already-included module. When such a module is found, it is included in the program being built.

In other words, when scanning a library of object modules, the linker only includes needed modules in the program it is building.

A filename passed to the linker has the standard AmigaDos path format, i.e., it consists of an optional volume name, optional path to the directory containing the file, and the filename itself. The volume defaults to the current volume and the directory to the current directory.

EXECUTABLE FILES

The name of the file to which the linker writes the executable program can be specified using linker option `-o`. If this option is not used, the linker derives the name of the executable file from that of the first object module listed on the command line, by deleting the extension. In the default case, the executable file is placed in the same directory in which the first object module is located.

For example, the following instruction links `main.o`, `menu.o`, and `add.o` with the Manx library `c.lib`, all of which are in the current

directory, and sends the executable program to the file named **main** in the current directory:

```
ln main.o menu.o add.o c.lib
```

And the following links the same modules, placing the output in **sys:bin/myprog**

```
ln -o sys:bin/myprog main.o menu.o add.o c.lib
```

LIBRARIES

Aztec C provides the following basic libraries:

c.lib
m.lib

All programs must be linked with **c.lib**. In addition to all the nonfloating point functions described in the library chapter, **c.lib** contains internal functions that are called by compiler-generated code, such as functions to process switch statements.

m.lib contains the transcendental floating point functions described in the library chapter, such as **sin()**, and versions of the functions **printf()**, **fprintf()**, **sprintf()**, **scanf()**, **fscanf()**, and **sscanf()**. A program that calls any of these functions must be linked with **m.lib** in addition to **c.lib**.

Otherwise, a program need not be linked with **m.lib**. In particular, if it performs floating point operations without calling any of these functions, it does not need to be linked with **m.lib**.

When a program calls a **printf()**- or **scanf()**-type function to perform a floating point conversion, the linker must search **m.lib** before it searches **c.lib**. The reason for this is that there are two versions of these functions: the ones that support floating point are in **m.lib**; the others that do not are in **c.lib**. If **c.lib** is searched before **m.lib**, then when a program that requires the floating point versions is linked, the nonfloating point version of a function will be used and the program will misbehave.

For additional information on the other libraries included with your Aztec C development package refer to the **Library Overview** chapter of the *Aztec C Library Manual*.

SUMMARY OF LINKER OPTIONS

The linker options are summarized as follows:

- | | |
|----------------|---|
| +a | Forces each module to be aligned on a long word boundary. For most applications this is not necessary and will only make the program larger. However, in certain cases (i.e. BCPL pointers) it is necessary for long word alignment. |
| +c[cdb] | Forces loading into chip memory (i.e. standard built-in memory) of the program sections whose identifying letters follow the +c. |
| +f | Forces loading into fast memory (i.e. external add-on memory) of the program sections whose identifying letters follows the +f. |
| -f file | Reads command arguments from <i>file</i> . |
| -g | Tells the linker to generate a source level debugger output file. This file has the same root name as the program output file and the extension .dbg . This file is used by the optional Aztec C Source Level Debugger. |
| +l | This option tells the linker that the following Amiga object modules are to be treated as though they are libraries until the next +l is encountered. This option acts as a toggle and may be used several times in a single link line. |
| -l name | Library, e.g., -lc for c.lib |
| -m | Disables linker warnings about object module symbols overriding library symbols. |

- o *file*** Specifies the name *file* to which the executable program will be sent. If not given, the name of the file is the same as that of the first input file, with the extension deleted.
- +o*i*** Places the following object modules in code segment *i*. If no number is specified, use the first empty segment. If the segment already exists, the modules are added to its end.
- q** Turns off source level debugging file generation. (See **-g**.)
- +q** Disable the module by module display while linking. The linker displays the names of the various modules as it reads them. This option causes the linker to suppress the display.
- +sss +ss+s *val*** Specifies four different models for scatter loading, ranging from one big hunk to every module in its own hunk.
- t** Generates an ASCII symbol table file.
- w** Generates a Wack-readable symbol table. This table is also usable by db (see Chapter 7, **DB Debugger**.)
- v** Specifies verbose link.

DETAILED DESCRIPTION OF OPTIONS

Option -l

Option **-l** provides a convenient way to link programs located in one directory with libraries located in another.

Option **-l** is immediately followed by the partial name of a library file. The linker builds the complete name by prefixing it with the string associated with the environment variable **CLIB** and appending the string **.lib** to it.

For example, if the libraries are located in the directory `sys:lib`, then set CLIB to:

```
sys:lib/
```

and the object module `prog.o` could be linked with the libraries `mylib.lib`, `m.lib`, and `c.lib` using the command

```
ln prog.o -lmylib -lm -lc
```

Option -f

This option causes the linker to continue reading options filename from a file; when done, it then continues reading arguments from the command line. The name of the file follows option -f.

For example, the following command links `prog.o` with `sub1.o`, `sub2.o`, ..., `m.lib` and `c.lib`; it reads some filenames from the file `prog.lnk`:

```
ln -f prog.lnk -lm -lc
```

where `prog.lnk` contains

```
-o prog.out  
sub1.o sub2.o  
sub3.o  
sub4.o
```

The contents of files read by the -f option should contain only the names of object modules and libraries to be linked. It should not contain any additional linker options.

Option +l

This flag specifies that the following Amiga object modules are really Amiga libraries until the next option +l is detected. The Linker automatically detects that a module is in Amiga format. However, because an Amiga object library is simply a concatenation of a number of modules, it is necessary to tell the Linker that the following module is a library.

Otherwise, the Linker adds all the modules in the file to the program. For example, if modules **am1.o** and **am2.o** are Amiga libraries, they can be linked in as follows along with the program **prog.o**:

```
ln prog.o +l am1.o am2.o +l -lc
```

Option -m

Normally, when you create a variable in your program that has the same name as a library routine, the linker issues a warning that your symbol will override the library symbol. If the **-m** option is specified, the warnings are suppressed.

Option -o

Option **-o** can be used to specify the name of the file to which the linker is to write the executable program. The name of this file is in the parameter that follows the **-o**. For example, the following command writes the executable program to the file **progout**:

```
ln -o progout prog.o c.lib
```

If this option is not used, the linker derives the name of the executable file from that of the first input file, by deleting its extension.

Option +o[i]

The linker uses this option to handle segmentation (overlay). The executable code in the object modules that follow the option is placed in code segment **i**. If **i** is not specified, use the first empty segment number. If the segment already exists, append the code to its end. When segments are used, the linker generates a reference to the symbol **.segload** which is defined in a library module, **segload.o**. This module must be in the root segment for the program to function properly. The module is also available directly in the **lib** directory.

Segments are loaded into memory as needed and remain in memory until explicitly removed by the program. The program does this by calling the **freeseq()** routine with the address of a function that is in the segment to be unloaded.

Option +s, +ss, +sss

The Linker supports four models of scatter loading. Default model is no option and specifies code in one hunk. Linker option **+s** places each file in a separate hunk. This means that modules concatenated together into a single file and all modules pulled from a single library are placed together in the same hunk. Linker option **+ss** adds files to a hunk until the size of the hunk reaches 8K. The linker then starts a new hunk. Finally, Linker option **+sss** puts each module into its own individual hunk.

The Linker generates a **jmp** instruction at the beginning of Hunk 0 if there has been an entry point defined and the entry point is not already at the beginning of Hunk 0.

Option -t

The **-t** option creates an ASCII file that contains a symbol table for the linker. This file is just a text file which lists each symbol with a hexadecimal address. This address is either the entry point for a function or the location in memory of a data item. A perusal of this file will indicate which functions were actually included in the program.

The symbol table file will have the same name as that of the file containing the executable program, with extension changed to **.sym**.

There are several special symbols which will appear in the table. They are defined in the **Program Organization** section of the **Technical Information** chapter.

Option -w

The **-w** option creates a disk file that contains a symbol table for the linker, in the format expected by Wack.

Option -v

The **-v** option causes the linker to send a progress report of the linkage to the screen as each input file is processed. This is useful in tracking down undefined symbols and other errors which may occur while linking.

Options +c and +f

There are two types of memory on an Amiga: 'chip' memory, which is inside the Amiga; and 'fast' memory, which is external to the Amiga. By default, the three sections of a program (code, initialized data, and uninitialized data) are loaded into whatever memory is available. The +c and +f options allow you to force selected sections of a program to be loaded into chip and fast memory, respectively. Program sections are identified to an option by appending a letter to the option. These letters, and the sections to which they refer are:

- c** Executable code.
- d** Initialized data.
- b** Uninitialized data (bss).

If an option is used without appending any letters to it, then all the program's sections are forced to be loaded into the specified type of memory.

For example:

```
ln +cdb +fc ....
```

will force a program's initialized and uninitialized data to be loaded into chip memory and the program's executable code to be loaded into fast memory. Note that if there is no extra memory, the program will not load.

More normally, you would use:

```
ln +cdb ....
```

which will place the program's initialized and uninitialized data into chip memory and its executable code into whatever memory is available.

Finally:

```
ln +c ...
```

will force the program's initialized data, uninitialized data, and executable code to be loaded into chip memory; it is equivalent to +ccdb.

Option -g

The **-g** option is for use with the Manx Source Level Debugger (**sdb**). This option, combined with the compiler's **-bs** option, will generate a file with the **.dbg** suffix in the current directory. This **.dbg** file provides information necessary for use with **sdb**.

For example:

To compile and link the "hello, world" program and generate the **.dbg** file, you would use the following commands:

```
cc -bs hello.c  
ln -g hello.o -lc
```

This will create the file **hello.dbg** in the current directory.

Using the **-g** option without having SDB installed will have no effect on the final program execution.

UTILITIES

6

Chapter 6 - Utilities

This chapter describes the Aztec C utility programs, in alphabetical order.

NAME	<code>arcv</code> - text dearchiver
SYNOPSIS	<code>arcv <i>arcvfile</i> [<i>dest_dir</i>]</code>
DESCRIPTION	<code>arcv</code> is the counterpart to the utility <code>mkarcv</code> and is designed to extract files that were originally archived with <code>mkarcv</code> . <code>arcv</code> will take the archive <i>arcvfile</i> and extract all of the text files stored within it. The original archived file is left intact. The extracted files are placed in the directory specified by <i>dest_dir</i> ; if <i>dest_dir</i> is not specified, the files are placed in the current directory. If subdirectories have been included in the archive, <code>arcv</code> will create these and place the dearchived text files in them.
EXAMPLE	For example, if you have an archived file named <code>example.arc</code> which contains two files, <code>explore.c</code> and <code>makefile</code>, then the command: <pre>arcv example.arc</pre> will place the files <code>explore.c</code> and <code>makefile</code> in the current directory.

NAME `adump` - Amiga object module dump utility

SYNOPSIS `adump [-cds] file`

DESCRIPTION `adump` displays information about *file*, which contains an object module or load module that has been generated by the Amiga Assembler or Linker.

`adump` always lists the length of each of the module's blocks and the offset of each block. The options to `adump` cause it to display the following additional information:

Option	Meaning
-c	Display code block contents in hex
-d	Display data block contents in hex
-s	Display symbol block contents (i.e., symbol names). Symbol blocks are found in executable files if debugging information has been included.

NAME	<code>cmp</code> - file comparison utility
SYNOPSIS	<code>cmp [-l] file1 file2</code>
DESCRIPTION	<code>cmp</code> compares two files on a character-by-character basis. When it finds a difference, it displays a message, giving the offset from the beginning of the file. If option <code>-l</code> is not specified, the program will stop after the first difference, displaying a message in the format: Files differ: character 10 If option <code>-l</code> is specified, <code>cmp</code> lists all differences, in the format: <i>decimal-offset hex-offset: f1-value f2-value</i> where <i>f1</i> and <i>f2</i> are files.
EXAMPLES	<pre>cmp otst ntst Files differ: character 10 and cmp -l otst ntst 10 a: 00 45 100 64: 1a 23</pre>

NAME	cnm - display object file info
SYNOPSIS	cnm [-sol] <i>file</i> [<i>file</i> ...]
DESCRIPTION	cnm displays the size and symbols of its object file arguments. The files can be object modules created by the Manx assembler or libraries of object modules created by the lb librarian. However, cnm does not support Amiga object module format. For example, the following displays the size and symbols for the object module sub1.o and the library c.lib: <pre>cnm sub1.o c.lib</pre> By default, the information is sent to the console. It can be redirected to a file or device in the normal way. For example, the following commands send information about sub1.o to the display and to the file dispfile: <pre>cnm sub1.o cnm > dispfile sub1.o</pre> The first line listed by cnm for an object module has the following format: <pre><i>file</i> (<i>module</i>): code: <i>cc</i> data: <i>dd</i> udata: <i>uu</i> total: <i>tt</i> (0xhh)</pre> where • <i>file</i> is the name of the file containing the module • <i>module</i> is the name of the module. If the module is unnamed, this field and its surrounding parentheses are not printed • <i>cc</i> is the number of bytes in the module code segment, in decimal • <i>dd</i> is the number of bytes in the modules' initialized data segment, in decimal • <i>uu</i> is the number of bytes in the module uninitialized data segment, in decimal

- *tt* is the total number of bytes in the module's three segments, in decimal
- *hh* is the total number of bytes in the module's three segments, in hexadecimal.

OPTIONS

Option List

- s Display size only
- l Display each symbol on a separate line
- o Prefix symbols with filename

If **cnm** displays information about more than one module, it displays four totals just before it finishes, listing the sum of the sizes of the module code segments, initialized data segments, and uninitialized data segments, and the sum of the sizes of all segments of all modules. Each sum is in decimal; the total of all segments is also given in hexadecimal.

Option **-s** tells **cnm** to display only the sizes of the object modules. If this option is not specified, **cnm** also displays information about each named symbol in the object modules.

When **cnm** displays information about the module named symbols, option **-l** tells **cnm** to display each symbol's information on a separate line and to display all of the characters in a symbol's name; if this option is not used, **cnm** displays the information about several symbols on a line and only displays the first eight characters of a symbol name.

Option **-o** tells **cnm** to prefix each line generated for an object module with the name of the file containing the module and the module name in parentheses (if the module is named). If this option is not specified, this information is listed just once for each module, prefixed to the first line generated for the module.

Option **-o** is useful when using **cnm** in combination with **grep**. For example, the following commands display all information about the module **perror** in the library **c.lib**:

```
cnm -o c.lib tmp
grep perror tmp
```

Symbol Format

cnm displays information about a module's "named" symbols, e.g., about the symbols that begin with something other than a period followed by a digit. For example, the symbol *quad* is named, so information about it would be displayed; the symbol *.0123* is unnamed, so information about it would not be displayed.

For each named symbol in a module, **cnm** displays its name, a two-character code specifying its type, and an associated value. The value displayed depends on the type of the symbol.

Symbol Types

If the first character of a symbol type code is lowercase, the symbol can only be accessed by the module; in other words, the symbol is local to the module. If this character is uppercase, the symbol is global to the module: Either the module has defined the symbol and is allowing other modules to access it, or the module needs to access the symbol that must be defined as a global or public symbol in another module. The type codes are:

ab

The symbol was defined using the assembler's **equ** directive. The value listed is the equated value of its symbol.

The compiler does not generate symbols of this type.

pg

The symbol is in the code segment. The value is the offset of the symbol within the code segment.

The compiler generates this type symbol for function names. Static functions are local to the function, and so have type **pg**; all other functions are global, that is, callable from other programs, and hence have type **Pg**.

dt

The symbol is in the initialized data segment. The value is the offset of the symbol from the start of the data segment.

The compiler generates symbols of this type for initialized variables that are declared outside any function. Static variables are local to the program and so have type **dt**; all other variables are global, i.e., accessible from other programs, and hence have type **Dt**.

un

The symbol is used but not defined within the program. The value has no meaning.

In assembly language terms, a type of **Un** (the U is capitalized) indicates that the symbol is the operand of a public directive and that it is perhaps referenced in the operand field of some statements, but that the program did not create the symbol in a statement label field.

bs

The symbol is in the uninitialized data segment. The value is the space reserved for the symbol.

The compiler generates **bs** symbols for static, uninitialized variables that are declared outside all functions and that are not dimensionless arrays.

The assembler generates **bs** symbols for symbols defined using the **bss** assembler directive.

G1

The symbol is in the uninitialized data segment. The value is the space reserved for the symbol.

The compiler generates **G1** symbols for nonstatic, uninitialized variables that are declared outside all functions and that are not dimensionless arrays.

The assembler generates **G1** symbols for variables declared using the global directive that have a nonzero size.

NAME	du - disk usage
SYNOPSIS	du <i>directory1</i> [<i>directory2...</i>]
DESCRIPTION	The du command displays the disk usage information for the specified directory. The command automatically operates upon any subdirectories. The result is specified for each directory in blocks. The blocks from a subdirectory are added to the parent directory's total. If no argument is specified, it defaults to the current directory.

NAME	hd - hex dump utility
SYNOPSIS	hd [+ <i>n</i> [.]] <i>file1</i> [+ <i>n</i> [.]] <i>file 2</i> ...
DESCRIPTION	hd displays the contents of one or more files in hex and ASCII to its standard output. <i>file1</i>, <i>file2</i>, ... are the names of the files to be displayed. +<i>n</i> specifies the offset into the files where the display is to start and defaults to the beginning of the file. If +<i>n</i> is followed by a period, <i>n</i> is assumed to be a decimal number; otherwise, it is assumed to be hexadecimal. Each file will be displayed beginning at the last specified offset.
EXAMPLES	To display the data forks of the files oldtest and newtest, beginning at offset 0x16b and of the file named junk beginning at its first byte, you would use the command: <pre>hd +16b oldtest newtest +0 junk</pre> To display the contents of tstfil, beginning at byte 1000, you would use the command: <pre>hd +1000. tstfil</pre>

NAME	lb - object file librarian																								
SYNOPSIS	lb <i>library</i> [<i>options</i>] [<i>mod1 mod2 ...</i>]																								
DESCRIPTION	lb is a program that creates and manipulates libraries of object modules. The modules must be created by the Manx assembler.																								
ARGUMENTS AND OPTIONS	Library Argument lb acts upon a single library file. The first argument to lb (<i>library</i>, in the synopsis) is the name of this file. The filename extension for <i>library</i> is optional; if not specified, it is assumed to be .lib. Options Argument There are function code options and qualifier options. These options are summarized and then described in detail. FUNCTION CODE OPTIONS lb performs <i>one</i> function at a time on the specified library. The functions that lb can perform, and the corresponding option codes, are: <table><tr><th>Function</th><th>Code</th></tr><tr><td>create a library</td><td>(no code)</td></tr><tr><td>add modules to a library</td><td>-a,-i,-b</td></tr><tr><td>list library modules</td><td>-t</td></tr><tr><td>move modules in a library</td><td>-m</td></tr><tr><td>replace modules</td><td>-r</td></tr><tr><td>delete modules</td><td>-d</td></tr><tr><td>extract modules</td><td>-x</td></tr><tr><td>ensure module uniqueness</td><td>-u</td></tr><tr><td>define module extension</td><td>-e</td></tr><tr><td>read function from file</td><td>-f</td></tr><tr><td>help</td><td>-h</td></tr></table>	Function	Code	create a library	(no code)	add modules to a library	-a,-i,-b	list library modules	-t	move modules in a library	-m	replace modules	-r	delete modules	-d	extract modules	-x	ensure module uniqueness	-u	define module extension	-e	read function from file	-f	help	-h
Function	Code																								
create a library	(no code)																								
add modules to a library	-a,-i,-b																								
list library modules	-t																								
move modules in a library	-m																								
replace modules	-r																								
delete modules	-d																								
extract modules	-x																								
ensure module uniqueness	-u																								
define module extension	-e																								
read function from file	-f																								
help	-h																								

In the synopsis, the options argument is surrounded by square brackets. This indicates that the argument is optional; if a code is not specified, **lb** assumes that a library is to be created.

QUALIFIER OPTIONS

In addition to a function code, the options argument can optionally specify a qualifier that modifies **lb** behavior as it is performing the requested function. The qualifiers and their codes are:

verbose	-v
silent	-s

The qualifier can be included in the same argument as the function code, or as a separate argument. For example, to cause **lb** to append modules to a library, and be silent when doing so, any of the following option arguments could be specified:

```
-as
-sa
-a -s
-s -a
```

The mod Argument

The arguments *mod1*, *mod2*, etc. are the names of the object modules, or the files containing these modules, that **lb** is to use. For some functions, **lb** requires an object module name, and for others it requires the name of a file containing an object module. In the latter case, the file's extension is optional; if not specified, **lb** will assume that it is **.o**. You can explicitly define the default module extension using Option **-e** (see pages 6-22.)

Reading Arguments from Another File

lb has a special argument, **-f filename**, that causes it to read command line arguments from the specified file. When done, it continues reading arguments from the command line. Arguments can be read from more than one file, but the file specified

in a *-f filename* argument cannot itself contain a *-f filename* argument.

Basic Features of lb

This section describes the basic features of **lb**. The basic points you need to know about **lb** are:

- How to create a library
- How to list the names of modules in a library
- How modules get their names
- Order of modules in a library
- Getting **lb** arguments from a file.

HOW TO CREATE A LIBRARY

A library is created by starting **lb** with a command line that specifies the name of the library file to be created and the names of the files whose object modules are to be copied into the library. It does not contain a function code, and it is this absence of a function code that tells **lb** that it is to create a library.

For example, the following command creates the library **exmpl.lib**, copying into it the object modules that are in the files **obj1.o** and **obj2.o**:

```
lb exmpl.lib obj1.o obj2.o
```

Making use of the **lb** assumptions about filenames for which no extension is specified, the following command is equivalent to the above command:

```
lb exmpl obj1 obj2
```

An object module file from which modules are read into a new library can itself be a library created by **lb**. In this case, all the modules in the input library are copied into the new library.

When **lb** creates a library or modifies an existing library, it first creates a new library with a temporary name. If the function was successfully performed, **lb** erases the file having the same name as the specified library, and then renames the new library,

giving it the name of the specified library. Thus, **lb** makes sure it can create a library before erasing an existing one.

Note that there must be room on the disk for both the old library and the new.

HOW TO LIST THE NAMES OF MODULES IN A LIBRARY

To list the names of the modules in a library, use the **lb** option **-t**. For example, the following command lists the modules that are in **exmpl.lib**:

```
lb exmpl.lib -t
```

The list includes some ****DIR**** entries. These identify blocks within the library that contain control information. They are created and deleted automatically as needed, and cannot be changed by you.

HOW MODULES GET THEIR NAMES

When a module is copied into a library from a file containing a single object module (that is, from an object module generated by the Manx assembler), the name of the module within the library is derived from the name of the input file by deleting the input file's volume, path, and extension components.

For example, in the example given above, the names of the object modules in **exmpl.lib** are **obj1** and **obj2**.

An input file can itself be a library. In this case, a module's name in the new library is the same as its name in the input library.

ORDER OF MODULES IN A LIBRARY

The order of modules in a library is important, since the linker makes only a single pass through a library when it is searching for modules. For a discussion of this, see the **Introduction to Linking** section of the **Linker** chapter.

When **lb** creates a library, it places modules in the library in the order in which it reads them. Thus, in the example given above, the modules will be in the library in the following order:

```
obj1 obj2
```

As another example, suppose that the library `oldlib.lib` contains the following modules, in the order specified:

```
sub1 sub2 sub3
```

If the library `newlib.lib` is created with the command

```
lb newlib mod1 oldlib.lib mod2 mod3
```

the contents of the newly-created `newlib.lib` will be:

```
mod1 sub1 sub2 sub3 mod2 mod3
```

The `ord` utility program can be used to create a library whose modules are optimally sorted. For information, see the `ord` description later in this chapter.

GETTING LB ARGUMENTS FROM A FILE

For libraries containing many modules, it is frequently inconvenient, if not impossible, to enter all the arguments to `lb` on a single command line. In this case, the `lb -f filename` feature can be of use: when `lb` finds this option, it opens the specified file and starts reading command arguments from it. After finishing the file, it continues to scan the command line.

For example, suppose the file `build` contains the line

```
exmpl obj1 obj2
```

Then entering the command

```
lb -f build
```

causes `lb` to get its arguments from the file `build`, which causes `lb` to create the library `exmpl.lib` containing `obj1` and `obj2`.

Arguments in a `-f` file can be separated by any sequence of whitespace characters ('whitespace' being blanks, tabs, and newlines). Thus, arguments in a `-f` file can be on separate lines, if desired.

The `lb` command line can contain multiple `-f` arguments, allowing `lb` arguments to be read from several files. For example,

if some of the object modules that are to be placed in **exmpl.lib** are defined in **arith.inc**, **input.inc**, and **output.inc**, then the following command could be used to create **exmpl.lib**:

```
lb exmpl -f arith.inc -f input.inc  
-f output.inc
```

A **-f** file can contain any valid **lb** argument, except for another **-f**. That is, **-f** files cannot be nested.

ADVANCED lb FEATURES

In this section we describe the rest of the functions that **lb** can perform. These primarily involve manipulating selected modules within a library.

ADDING MODULES TO A LIBRARY

lb allows you to add modules to an existing library. The modules can be added before or after a specified module in the library or can be added to the beginning or end of the library.

The options that select the **add** function are:

Option	Function
-b target	add modules before the module target
-i target	same as -b target
-a target	add modules after the module target
-b+	add modules to the beginning of the library
-i	same as -b+
-a+	add modules to the end of the library

In an **lb** command that selects the **add** function, the names of the files containing modules to be added follows the **add** option code (and the target module name, when appropriate). A file can contain a single module or a library of modules.

Modules are added in the order that they are specified. If a library is to be added, its modules are added in the order they occur in the input library.

ADDING MODULES BEFORE AN EXISTING MODULE

As an example of the addition of modules before a selected module, suppose that the library `exmp1.lib` contains the modules

```
obj1  obj2  obj3
```

The command

```
lb exmp1 -i obj2 mod1 mod2
```

adds the modules in the files `mod1.o` and `mod2.o` to `exmp1.lib`, placing them before the module `obj2`. The resultant `exmp1.lib` looks like this:

```
obj1  mod1  mod2  obj2  obj3
```

ADDING MODULES AT THE BEGINNING OR END OF A LIBRARY

Options `-b+` and `-a+` tell `lb` to add the modules whose names follow the option to the beginning or end of a library, respectively. Unlike options `-i` and `-a`, these options are not followed by the name of an existing module in the library.

For example, given the library `exmp1.lib` containing

```
obj1  obj2
```

the following command will add the modules `mod1` and `mod2` to the beginning of `exmp1.lib`:

```
lb exmp1 -i+ mod1 mod2
```

resulting in `exmp1.lib` containing

```
mod1  mod2  obj1  obj2
```

The following command will add the same modules to the end of the library:

```

lb exmp1 -a+ mod1 mod2
resulting in exmp1.lib containing

obj1  obj2  mod1  mod2

```

MOVING MODULES IN A LIBRARY

Modules which already exist in a library can be easily moved about, using the move option, **-m**.

As with the options for adding modules to an existing library, there are several forms of move functions:

Option	Meaning
-mb <i>target</i>	move modules before the module <i>target</i>
-ma <i>target</i>	move modules after the module <i>target</i>
-mb+	move modules to the beginning of the library
-ma+	move modules to the end of the library

In the **lb** command, the names of the modules to be moved follow the move option code.

The modules are moved in the order in which they are found in the original library, not in the order in which they are listed in the **lb** command.

DELETING MODULES

Modules can be deleted from a library using option **-d**. The command for deletion has the form

```
lb libname -d mod1 mod2 ...
```

where *mod1*, *mod2*, ... are the names of the modules to be deleted.

For example, suppose that **exmp1.lib** contains

```
obj1  obj2  obj3  obj4  obj5  obj6fP
```

The following command deletes **obj3** and **obj5** from this library:

```
lb exmp1 -d obj3 obj5
```

REPLACING MODULES

The replace option is used to replace one module in a library with one or more other modules.

The replace option has the form `-r target`, where *target* is the name of the module being replaced. In a command that uses the replace option, the names of the files whose modules are to replace the target module follow the replace option and its associated target module. Such a file can contain a single module or a library of modules.

Thus, an `lb` command to replace a module has the form:

```
lb library -r target mod1 mod2 ...
```

For example, suppose that the library `exmp1.lib` looks like this:

```
obj1 obj2 obj3 obj4
```

Then to replace `obj3` with the modules in the files `mod1.o` and `mod2.o`, the following command could be used:

```
lb exmp1 -r obj3 mod1 mod2
```

resulting in `exmp1.lib` containing

```
obj1 obj2 mod1 mod2 obj4
```

UNIQUENESS

`lb` allows libraries to be created containing duplicate modules, where one module is a duplicate of another if it has the same name.

Option `-u` causes `lb` to delete duplicate modules in a library, resulting in a library in which each module name is unique. In particular, option `-u` causes `lb` to scan through a library, looking at module names. Any modules found that are duplicates of previous modules are deleted.

For example, suppose that the library `exmp1.lib` contains the following:

```
obj1 obj2 obj3 obj1 obj3
```


The command

```
lb exmpl -u
```

will delete the second copies of the modules **obj1** and **obj2**, leaving the library looking like this:

```
obj1 obj2 obj3
```

EXTRACTING MODULES

Extracting modules from a Library option **-x** extracts modules from a library and puts them in separate files, without modifying the library.

The names of the modules to be extracted follow the **-x** option. If no modules are specified, all modules in the library are extracted.

When a module is extracted, it is written to a new file; the file has same name as the module and extension **.o**.

For example, given the library **exmpl.lib** containing the modules

```
obj1 obj2 obj3
```

the command

```
lb exmpl -x
```

extracts all modules from the library, writing **obj1** to **obj1.o**, **obj2** to **obj2.o**, and **obj3** to **obj3.o**.

And the command

```
lb exmpl -x obj2
```

extracts just **obj2** from the library.

THE VERBOSE OPTION

The verbose option, **-v**, causes **lb** to be verbose; that is, to tell you what it is doing.

SILENCE OPTION

The silence option, **-s**, tells **lb** not to display its sign on message. This option is especially useful when redirecting the output of a list command to a disk file.

REBUILDING A LIBRARY

The following commands provide a convenient way to rebuild a library:

```
lb exmpl -st > tfil  
lb exmpl -f tfil
```

The first command writes the names of the modules in **exmpl.lib** to the file **tfil**. The second command then rebuilds the library, using as arguments the listing generated by the first command.

The **-s** option to the first command prevents **lb** from sending information to **tfil** that would foul up the second command. The names sent to **tfil** include entries for the directory blocks, ****DIR****, but these are ignored by **lb**.

DEFINING THE DEFAULT MODULE EXTENSION

Specification of the extension of an object module file is optional; **lb** assures that the extension is **.o**. You can explicitly define the default extension using option **-e**. This option has the form

```
-e .ext
```

For example, the following command creates a library; the extension of the input object module files is **.i**.

```
lb my.lib -e .i mod1 mod2 m
```

HELP

Option **-h** will generate a summary of **lb** functions and options.

NAME	<code>ls</code> - list files and directories				
SYNOPSIS	<code>ls [-blprst] [file_pattern directory...]</code>				
DESCRIPTION	<code>ls</code> lists the names of the files and the contents of the directories whose names are passed to it as arguments. The files you wish to list should be specified by one or more <i>file_pattern</i> arguments. <i>file_pattern</i> may be a <i>filename</i> with or without a path specification, or a <i>filename</i> mixed with wild card characters. The valid wild card characters are: '?' - matches exactly one unknown or variable character. '*' - matches an arbitrary number of unknown characters. Wildcards are used to list multiple file names that are related in some way. For example, <pre>ls *.c</pre> will list all files with a .c extension. <pre>ls ? bc.c</pre> will list filenames such as <code>abc.c</code>, <code>bbc.c</code>, and <code>lbc.c</code>, but not <code>aabc.c</code> If no names are specified, the contents of the current directory are listed. The information is written to the <code>ls</code> standard output device, and thus goes, by default, to the screen. The information can be sent to another device or file by redirecting the <code>ls</code> standard output device in the normal manner. By default, the output is sorted alphabetically in multiple columns with directories displayed in an offsetting color. When directed to a file, the output is always alphabetical, one name to a line. <table><thead><tr><th>Options</th><th>Meaning</th></tr></thead><tbody><tr><td><code>-b</code></td><td>Displays the number of blocks in the file in addition to any other information requested.</td></tr></tbody></table>	Options	Meaning	<code>-b</code>	Displays the number of blocks in the file in addition to any other information requested.
Options	Meaning				
<code>-b</code>	Displays the number of blocks in the file in addition to any other information requested.				

- l Generates additional information for each file listed, including attributes, size, and last modification date
- p Forces a blank to be put before each filename and a - before each directory.
- r Reverses the order that has been specified.
- s Causes the list to be sorted by size, with the largest files first.
- t Causes the list to be sorted in chronological order with the most recent files first.

NAME	mkarcv - text file archiver
SYNOPSIS	mkarcv <i>arcfile</i>
DESCRIPTION	mkarcv combines several individual text files into one "archive" file, where the parameter <i>arcfile</i> is the name of the archive file. mkarcv reads the names of the files to be archived from its standard input device. Each filename is on a separate line. Usually, a separate file containing the names of the files to be archived is created first and mkarcv's standard input is redirected to this file in the standard manner. Alternatively, you can type in each file's name from the console, separating each name with a carriage return and ending the list with a period as the first character on a line by itself. Files archived with mkarcv may be dearchived with the arcv utility described elsewhere in this chapter. Please note that files archived with mkarcv are not in any way altered or "squeezed," and the archive can be edited with a standard text editor.
EXAMPLE	To archive the files explore.c and makefile into the file example.arc, you would create a file named input.txt containing these two filenames with a carriage return after each name: <pre>explore.c <CR> makefile <CR></pre> and then redirect the standard input of mkarcv to this file: <pre>mkarcv <input.txt example.arc</pre> You can also interactively enter the filenames when you run the mkarcv command: <pre>mkarcv example.arc <CR> explore.c <CR> makefile <CR> . <cr></pre>

NAME	<code>obd</code> - list object code
SYNOPSIS	<code>obd objfile [,objfile,...]</code>
DESCRIPTION	<code>obd</code> lists the loader items in an object file. <i>objfile</i> is the Aztec object module which you desire to examine.

NAME	ord - sort object module list
SYNOPSIS	ord [-v] [<i>infile</i> [<i>outfile</i>]]
DESCRIPTION	ord sorts a list of object filenames for use by the lb utility. A library that is generated from this sorted list will contain a limited number of backward references, i.e., global symbols that are defined in one module and referenced in a later module. Since the specification of a library to the linker causes it to search the library only once, a library having no backward references must be specified only once when linking a program, and a library having backward references may need to be specified multiple times. <i>infile</i> is the name of a file containing an unordered list of filenames. These files contain the object modules that are to be put into a library. If <i>infile</i> is not specified, this list is read from the ord standard input. The filenames can be separated by space, tab, or newline characters. <i>outfile</i> is the name of the file to which the sorted list is written. If it is not specified, the list is written to the ord standard output. <i>outfile</i> can only be specified if <i>infile</i> is also specified. Option -v causes ord to be verbose, sending messages to its standard error device as it proceeds.

NAME	<code>set</code> - environment variable and exec file utility
SYNOPSIS	<code>set</code> <code>set VAR=string</code>
DESCRIPTION	<code>set</code> is used to examine and set environment variables. Displaying and Setting Environment Variables The first form listed for <code>set</code> causes it to display the name and value of each environment variable. The second form assigns <code>string</code> to the environment variable <code>VAR</code>. In the second form, if <code>string</code> is not given, the specified variable is deleted from the environment.
SEE ALSO	See the Tutorial chapter of your <i>Aztec Shell User Guide</i> for more information on environment variables.

NAME	setdate - set date and time
SYNOPSIS	setdate
DESCRIPTION	setdate allows you to set the date and time. It prompts you as follows: Date (MM/DD/YY) ? Time (HH:MM:SS) ? If you type 02/03, it leaves the year alone. If you type 12 in response to the Time prompt, it assumes 0 for the minutes and seconds.

DEBUGGER

7

Chapter 7 - Debugger

db is a symbolic debugger. It is used to debug programs which have been created using the Aztec C compiler, assembler, and linker.

db has all the standard features of an assembly language debugger. It also has features not found in all debuggers, such as the ability to reference memory locations by name as well as by address, the ability to define sequences of commands to be macros, which can then be activated by entering a single letter, and a flexible mechanism for handling breakpoints.

In addition, **db** has features specifically tailored to its use with Aztec C, such as the ability to list the name and parameters of the currently executing function, and the function that called it, and so on, back to the initial function. Another special feature is the ability to display, on entry and exit from each function, the function's parameters and return value. Finally, **db** supports both scatter-loaded and segmented programs.

Requirements

An Amiga with at least 512k of memory is recommended for use with **db**. The debugger itself uses about 96k.

Preview

This chapter is divided into three sections: **Overview**, which describes **db** features in more detail and introduces the commands; **Using db**, which describes in full detail how to use **db**; and a **Command Summary**.

Overview

db commands consist of one or two characters, the first of which identifies the command category. If there is only one command in the category, then the command has just one letter; otherwise, the command has a second letter which identifies the specific operation to be performed.

BASIC COMMANDS

db has two types of commands for examining memory: display and print, whose first characters are **d** and **p**, respectively. The **DISPLAY** commands **db** and **dw** simply display hexadecimal bytes and words.

The **PRINT** command, **p**, is more powerful, being able to convert a sequence of one or more possibly different types of data items to ASCII. For example, you can tell the print command that beginning at the location **var** is a sequence of the following items: an **int**, a **float**, and a pointer to a **char** string. The **p** command will convert the two binary items to ASCII and print them, then display the referenced character string.

The **REGISTER** command, **r**, displays and modifies the 68000 registers.

The **MEMORY MODIFY** commands, **m**, modify memory.

The **u** commands **UNASSEMBLE** code; that is, they display it symbolically, in a form similar to its appearance in an assembly language source file.

The **s**, **t** and **g** commands cause your program to be executed. **s** and **t** commands **SINGLE STEP** your program; that is, they execute a specified number of instructions in your program and then return control to **db**. The **t** command differs from the **s** command by single stepping over **jsr** and **bsr** instructions. **g** commands transfer control of the processor unconditionally to your program. In this case, **db** regains control when your program terminates, when an error occurs (such as division by zero), or when a "breakpoint" is taken. Breakpoints are discussed later in this chapter.

? is the **HELP** command: It causes **db** to display a summary of all **db** commands. For some command categories, you can get information about the commands in a category by typing the first letter of the category's commands followed by a ?. For example, typing **m?** gets you information about the memory modification commands (all of whose first letter is **m**).

NAMES

db allows memory locations to be referenced by name as well as by location. It learns a program's global names by reading the symbols from the program file and placing them in a memory-resident symbol table. The linker generates a symbol table hunk for a program in response to the **-w** option.

db only allows global symbols to be accessed by name; automatic variables and static variables cannot be accessed by name.

You can also define names to **db** using the **v** command; The **CLEAR SYMBOLS** command, **cs**, will remove symbols from the memory-resident symbol table.

Code and Data symbols.

db classifies symbols as being either code or data symbols. All symbols in the program's symbol table hunks which occur between the special linker symbols **_H1_org_** and **_H1_end_** and between **_H2_org_** and **_H2_end_** are considered to be data symbols, and all others are code symbols.

There are two commands for viewing the symbols which are known to **db**: **dc** and **dd**, which display code and data symbols, respectively.

Operator usage of names.

When a C source program is compiled, all global names are prepended with an underscore character. To refer to symbols within **db**, the name can be typed without the underscore. First, the name will be searched for exactly as typed. If not found, **db** will prepend an underscore and search again.

Note that this means to reference a symbol such as **_main**, you would have to type **__main** to differentiate it from the **main** symbol, since typing **_main** would match before the extra underscore is prepended.

LOADING PROGRAMS AND SYMBOLS

The **db** program is started independently of the program to be debugged. The **al** command causes the debugger to wait till a program is loaded either from the Workbench or from the CLI; the program is stopped before it

executes any instructions. The debugger will also load the symbols from the program at this time if the program is in the current directory. If not in the current directory, the `LOAD SYMBOLS` command, `ls`, can be used.

Unfortunately, the Amiga operating system contains no mechanism for terminating a program from an external process. So, there are two ways for the debugger to terminate the program. The first method is to simply allow the program to resume using the `ar` command and let it terminate normally. If the program is unlikely to terminate normally on its own, the `ak` command checks the symbol table for the `_abort` symbol and then the `exit` symbol and sets the program counter to the first one found, allowing the program to resume.

BREAKPOINTS

Before transferring control of the processor to your program in response to a `g` command, `db` can set **BREAKPOINTS** at specified locations in the code. When your program reaches a breakpoint, `db` regains control.

A breakpoint has a **SKIP COUNT** associated with it, which allows a breakpoint to be passed several times before actually taking the breakpoint and returning control to `db`. When a breakpoint is reached, `db` is always activated; it increments a counter associated with the breakpoint. When the counter's value is greater than the breakpoint's skip count, the breakpoint is taken; that is, `db` retains control of the processor. Otherwise, `db` returns control of the processor to your program after the breakpoint. By default, a breakpoint's skip count is 0; thus, each time the breakpoint is reached, it is taken.

A breakpoint can also have a sequence of `db` commands associated with it. When a breakpoint is taken, these commands will be executed before `db` allows you to enter commands. For example, if you just want to examine a variable each time a certain location in the code is reached and then have the program continue execution, you could define a breakpoint at the location, and specify a list of commands to do just that: the first command in the sequence would be a `d` command to display memory, and the second would be a `g` command to continue execution of the program.

There are two ways to define breakpoints: with the `g` command, and with special breakpoint commands, whose first letter is `b`.

The breakpoint commands manipulate a table of breakpoints: there are commands for entering breakpoints into the table, displaying the entries, resetting their counters, and removing them from the table.

There is a difference between a breakpoint defined in a **g** command and those in the breakpoint table: The **g** command breakpoint is temporary, while a table breakpoint is more permanent (it exists until removed from the table). Before transferring control to your program in response to a **g** command, **db** sets all breakpoints that are in the breakpoint table and that are specified in the **g** command itself. When a breakpoint is taken, **db** removes all breakpoints from the code and forgets all about the **g** command breakpoint. The table breakpoints, however, are still in the table and will be set back in memory when control is again returned to your program.

db remembers the skip counter associated with a breakpoint which is in the breakpoint table: When it sets breakpoints in memory, the count for such a breakpoint is set to its remembered value (that is, its value in the table); and when a breakpoint is taken, the accumulated count for the breakpoints in memory are saved in the breakpoint table.

MEMORY-CHANGE BREAKPOINTS

The breakpoints described above are taken when a program reaches a specified point in the code. A second type of breakpoint, called a **MEMORY-CHANGE BREAKPOINT**, is taken when a specified memory location is changed from or set to a particular value.

With a memory-change breakpoint set, **db** will detect either the function or the instruction which modifies the specified memory location, depending on whether your program was activated using a **g** command or is being single-stepped using an **s** or **t** command, respectively.

When your program is activated with a **g** command and a memory-change breakpoint is set, **db** will examine the specified memory location on entry to, and exit from, each function. It will take a breakpoint, that is, interrupt execution of the program and return control to the operator, when the contents of the memory location meets the specified condition.

When an **s** or **t** command is used to single-step a program and a memory-change breakpoint is set, **db** will examine the specified memory location after each instruction is executed, and take a breakpoint when appropriate.

The **bb**, **bw**, and **bl** commands are used to set and remove memory-change breakpoints.

TRACE MODE

db supports a **TRACE MODE**, which displays information whenever a function is entered or exited.

When entering a function with trace mode enabled, the function or trap name and its arguments are displayed. Optionally, upon exit from a function, a return value is displayed.

The commands **bt** and **bT** affect trace mode: **bt** enables and disables trace mode, and **bT** enables and disables the display of function exit information.

BACKTRACING

When **db** regains control from an executing program (for example, because a breakpoint was taken), it has the ability to display information on how the program got to its current location. The **ds** command will display information about the currently executing function, and the function which called it, and so on, back to the Manx function **_main**, which called your function **main**.

ds displays, for each function, its name, arguments which were passed to it, and the address to which it will return.

MACROS

db allows you to define and execute **MACROS**, that is, a sequence of **db** commands.

A macro is associated with a single alphabetical character, so up to 26 macros can be known to **db** at any time.

The **db** command **x** is used both to define and execute a macro.

OTHER FEATURES

Some other features of **db** are:

- The **EVALUATE EXPRESSION** command, **=**.
- The **HELP** command, **?**, which lists commands.
- The **INPUT RADIX** command, **n**, which changes the default radix for input and display.

Using DB

STARTING DB

db is started with a command of the form:

db

or by selecting the **DB** icon and clicking from the Workbench.

When **db** starts, it creates a window with a startup message and then performs several actions. First, if running under V1.2 or later, the window is created without activating it. Next, **db** will look in the current directory for a file with the name **.dbinit**. If found, **db** will open the file, read it, and execute commands from the file.

When **db** reaches the end of the file it checks to see whether it is waiting for a breakpoint or program load and if so skips the next step. If **db** is ready for a command, it will first look in the environment for a variable **DBINIT**. If found, **db** will open the file defined by this variable and read and execute commands from this file as well. In this way, it is possible to do system wide as well as local presets.

When **db** has finished reading either or both files it again looks to see if it is waiting for a breakpoint or a program load. If it is not, then **db** will activate the window and wait for input for the next command. If it is, then **db** will not activate the window.

For example, suppose you enter the following command,

```
set DBINIT=s:dbinit
```

then enter the following line in **s:dbinit**:

```
al
```

When you start **db**, it will open the window, read **s:dbinit** and wait for a program to load without activating the window. This means you can type:

```
db  
program
```

without having to use the mouse.

TERMINATION

If a program is a process or CLI program, **db** will set a breakpoint on the return address from the program. If this breakpoint is reached, **db** will print a message that the program has terminated normally and will automatically resume the task.

SUPPORT FOR SCATTER-LOADED PROGRAMS

This version of **db** contains support for scatter-loaded programs. Variables which are referenced using absolute addressing are displayed normally. Functions which are accessed through the jump table are displayed in parentheses. However, the names can be used as though not scatter-loaded.

SUPPORT FOR SEGMENTED PROGRAMS (OVERLAYS)

db supports breakpoints in overlays. This is not easy, so there are some caveats. First, any reference to a function in another segment will force the segment into memory. For example, saying:

```
u overl
```

where **overl** is a function will force the segment into memory before disassembling.

If breakpoints are set in the permanent breakpoint table, then when the **g** command is given, any breakpoints in non-resident segments will force the segments into memory before setting the breakpoint. (Think of it as a reference to that symbol.) Note that if a segment is unloaded by the program the breakpoint is lost even if the segment is reloaded by the program. However, if a segment is unloaded and the debugger regains control, the next **g** command will reload the segment and reset the breakpoint.

One potential problem to watch for: If you set a breakpoint, the program unloads the segment and the function gets called, but the breakpoint is no longer there.

INPUT/OUTPUT

While **db** is displaying information to the screen, the display can be stopped temporarily by typing **^S**. Any key will restart the display. Typing **^C** will abort the display.

When **db** is waiting for a breakpoint or program to load, typing any character will cause **db** to stop the current program. When **db** is displaying while running the program (Trace mode, for example) it is best to use **^C**, as other characters may be swallowed by the display routines.

COMMANDS

This section describes in detail the **db** commands. It first defines some terms that are used in the command descriptions.

Definitions

EXPR

An *expr* has the following form:

term [*binop term* ...]

That is, an *expr* can be a single *term* or a series of *terms* separated by binary operators. The binary operators are:

- +** addition
- subtraction
- *** multiplication

/	division
%	modulus
&	bitwise and
	bitwise inclusive or
^	bitwise exclusive or

All operators have the same precedence, and an unparenthesized *expr* is evaluated left to right. To override the default order of evaluation of an expression, you can parenthesize the relevant parts of the expression.

An *expr* has a 32-bit value. The operators that are applied to the *terms* out of which the *expr* is built affect just this 32-bit value.

TERM

A *term* always resolves to a numeric value, and can be one of the following:

<i>register</i>	<i>constant</i>
<i>-term</i>	<i>addr</i>
<i>*addr</i>	<i>#addr</i>
	<i>(expr)</i>

These names are defined in the following paragraphs.

- **REGISTERS** are specified by their standard names; *a0*, *d0*, *pc* and so on. The value of the *term* is the contents of the register.
- **CONSTANT** is a decimal, hexadecimal, octal number, or a character.

A sequence of digits preceded by *0x* is taken to be a hexadecimal number and those preceded by *0b* are binary numbers. A sequence of digits with a leading *0o* is taken to be an octal value. Digit strings ending in *.* are taken to be decimal values. If none of these prefixes or suffixes are present, the radix of the value is taken from the current radix. The default radix is hexadecimal. Note that if the current radix is set to hexadecimal, numbers must start with a digit to distinguish them from symbols (i.e. *fab*c will be taken to be a symbol where *0fab*c will be taken to be a hexadecimal number.) However, if a symbol is not matched, and all the letters are hex digits, the sym-

`bol` will be treated as a number. This is useful, but beware of something like:

```
dw foo+a0
```

because `a0` will use the value in register `a0` and not the value `0xa0`.

A character is represented by the character, surrounded by single quotes, as in `'x'`. The value of a character constant is its ASCII value.

Certain characters, the single quote `'` and the backslash `\` may also be defined within the single quotes. These are identified by a leading backslash character, and are:

char	hex value	db notation
newline	0a	\n
horizontal tab	09	\t
backspace	08	\b
carriage return	0d	\r
form feed	0c	\f
backslash	5c	\\
single quote	27	\'
bit pattern	<i>ddd</i>	\ddd

- **ADDR and *ADDR** When a *term* consists of a `*` followed by an *addr*, the value of the *term* is the contents of the 32-bit field referred to by the *addr*. For example:

- *var** The contents of the `var` field;
- *a3** The contents of the 32-bit field in the data segment pointed at by `a3`;
- *sp** The contents of the 32-bit field on the top of the stack;
- *(1b1+2)** The contents of the 32-bit field referred to by `1b1+2`;

Because an *addr* can itself be an *expr*, the **addr* term may require extra parentheses. For example:

***sp+2**

is equivalent to ***(sp+2)** and not **(*sp)+2**. The value of the first interpretation is the contents of the second word on the stack, while the value of the second is two plus the contents of the first word on the stack.

- **PERIOD (.)** The value of a *term* consisting of a period, ".", is the starting address *addr* of the last similar command. For example, if ten bytes of memory were displayed using the **db** command, as in:

db 0x100,10

then "." would be set to 0x100 for the next **db** or **dw** command. If the next **db** or **dw** command is

dw .

the same 10 bytes would be displayed as words.

"." has a separate value for the **u** command, for the **db**, **dw**, and **m** commands, and for the **p** command. An **m** command never modifies its associated ".".

- **@ [function]** symbol has as its value the return address of the specified function. The function name is optional, and defaults to the current function. The main use for @ is in the **g** command. For example:

g @

transfers control to your program, and sets a breakpoint at the return address of the current function.

As another example:

```
g @putc
```

transfers control to your program. When the function `putc` is reached, a breakpoint will be set at the address to which it will return.

ADDR

An *addr* defines the address of a location in memory, and has the form:

expr

Here are some examples of *addr*:

```
pc
main+10
.-40
*sp+8
data+*(a6+6)
```

RANGE

A *range* defines a block of memory. It has one of the following forms:

```
addr,cnt
addr>addr
addr
,cnt
```

The form *addr,cnt* specifies the starting address, *addr*, and a number, *cnt*. *cnt* is interpreted differently by different commands. For example, the `DISASSEMBLE CODE` command, `u`, will display *cnt* lines, while the `DISPLAY BYTES` command, `db`, will display *cnt* bytes.

The form *addr>addr* specifies the starting and ending addresses of *range*.

A full *range* need not be explicitly specified, because *db* remembers the last-used *range* and will set unspecified *range* parameters from the remembered values:

- When a *range* is specified which consists of a single *addr*, the last used *cnt* is used.
- When a *range* is specified which consists of *cnt*, the next consecutive address is used, and the remembered *cnt* is changed to the new value.
- When nothing is specified as the *range*, the next consecutive address is used as the starting *addr*, and the *cnt* is set to the remembered value.

CMDLIST

A *cmdlist* is a list of commands. It consists of a sequence of commands or macros separated by semicolons:

```
COMMAND [ ; COMMAND ... ]
```

If a macro is in a *cmdlist*, it must be the last command in the list.

COMMAND DESCRIPTIONS

The following descriptions of debugger commands use terms and concepts which were presented in the preceding sections.

The commands are listed alphabetically. For an index, see the command summary which follows the descriptions.

The Amiga Commands

- **add** display device list
- **adi** display interrupt list
- **adl** display library list
- **adp** display port list
- **adr** display resource list

Syntax:

add
adi
adl
adp
adr

Description:

These commands play the addresses and names of one of the various lists pointed to by ExecBase in the Amiga.

- **ai** display task information

Syntax:

ai

Description:

This command displays information about a particular task. If the task is a process or has been run from the CLI, additional information besides the task information is displayed.

If a task has already been selected, the ai command displays the information for the selected task. Otherwise, a list of tasks is displayed and the debugger prompts for you to enter a task.

- **ak** kill the current task

Syntax:

ak

Description:

This command is used to terminate the program or task currently being debugged. It does this by transferring control to the `_abort()` or `exit()` function of the program.

The Amiga operating system has no provisions for tracking memory or resource allocation. As a result, it is not possible to “kill” an executing task from an external one. Thus, if the program being debugged is unable to resume and exit normally, the system must be rebooted.

The only alternative is to force the program into its `exit()` function. This may or may not work depending upon the state of the program when the `ak` command is given. At the least, some memory is likely to be lost.

This command is made available as a shortcut for the equivalent command `r pc=exit`, and discretion in its use is advised.

- `al` debug the next program or task
- `aL` debug the next task without symbols

Syntax:

`al`
`aL`

Description:

Since the debugger operates as an independent task, it cannot load programs directly. Instead, it replaces certain vectors in the system and waits for the next program or task to be loaded. The debugger takes control before the task or program begins to execute.

Thus, to debug a program, this command is given and the debugger waits while you switch to a new window and either type a CLI command or click on a Workbench icon to start the program.

After the program is loaded and the debugger gains control, the symbols are loaded from the program file unless the `aL` command is used. Under 1.2, the window is automatically activated when the program is loaded.

- **an** create a new debugging window
- **aN** create a new debugging window with new symbols

Syntax:

an
aN

Description:

When the debugger is started, it creates its own window for commands and displays. With the **an** command it is possible to create additional windows for debugging more than one task at a time. If the **an** command is used, the new window will share symbols with the window active when the command was given. This is useful when debugging a sub-task that is a part of the active program. Note that the **al** command will not read symbols if a symbol table already exists.

When the **aN** command is used, the new window is created with an independent symbol table. This is useful for debugging several programs simultaneously.

- **am** memory information

Syntax:

am

Description:

This command displays the amount of free memory in the system.

- **ap** debug a crashed program (post-mortem)
- **aP** debug a crashed program without symbols

Syntax:

ap
aP

Description:

When a program run from the CLI or Workbench crashes because of an address or bus error or a divide by zero, etc., the trap is handled by a special DOS handler that puts up the "Software Failure" requester. Selecting **RETRY** has no effect, and selecting **CANCEL** passes control to **exec**'s trap handler which displays the **guru**. This function provides a way to examine the environment after the crash.

If **db** is running, or if you can start up **db** from another window, then using the **ap** command, **db** can act as though it was in control the whole time. When you give the command, **db** will ask you to select the task to be debugged. It will also make an informed guess as to which task it is. After selecting the task, **db** will instruct you to select the **CANCEL** button in the requester.

db will have control right at the point of error. At this point, it is possible to examine the state of things to determine why it happened and possibly to even recover by patching or skipping the code in error. Note that this is not always possible, since memory may have been corrupted before the error occurred.

Note also that if multiple requesters are present for different tasks, this will only work if you choose the most recent task that failed and click the proper requester.

► **aq** close all windows

Syntax:

aq

Description:

When more than one window is created using the **an** command, individual windows can be closed using the **q** command. The **aq** command will close all the windows and terminate the debugger. When the last window is closed, the debugger terminates.

When a window is closed, if a task was stopped for debugging, the task is allowed to resume.

The **aq** command performs a **q** command for each of the windows that is open.

- **ar** allow the current task to resume

Syntax:

ar

Description:

When a task has been selected for debugging, it is kept in a stopped state while commands are being given. The **ar** command deselects the task and allows **db** to resume. The window returns to the state it was in before a task was selected.

- **as** select a task to debug
- **aS** select a task without symbols

Syntax:

as
aS

Description:

It is possible for the debugger to debug a task or program that is already running in the system. The **as** command displays a list of all tasks in the system and prompts for the number of the task to debug. That task is stopped and the symbols for that task are loaded from the program file unless the **aS** form of the command is used.

- **at** display all tasks

Syntax:

at

Description:

The **at** command displays a list of all tasks in the system along with their priority, the address of the task block, the status and the name.

The Breakpoint Commands

- **bb** set byte memory-change breakpoint
- **bw** set word memory-change breakpoint
- **bl** set long memory-change breakpoint

Syntax:

```
bb
bw
bl
bb  addr [== [val] ]
bb  addr [!= [val] ]
bw  addr [== [val] ]
bw  addr [!= [val] ]
bl  addr [== [val] ]
bl  addr [!= [val] ]
```

Description:

These commands are used to set and clear a memory-change breakpoint, with the parameterized versions used to set breakpoints and the parameter-less versions to clear them. The **bb** command is used to monitor a one-byte field, the **bw** command to monitor a two-byte (word) field and the **bl** command to monitor a four-byte field.

In the parameterized form of the commands, *addr* specifies the field to be monitored.

With the **==** form, the breakpoint will be triggered when the debugger detects that the field is equal to the specified value, *val*.

With the **!=** form, the breakpoint will be triggered when the debugger detects that the field is different from the specified value.

The *val* parameter is optional. If not specified, **db** defaults to the current value at the *addr*. If the comparison field is also not specified, the default is **!=**. Thus, to break when a location is changed, use the command:

bw *addr*

- **bc** clear a single breakpoint
- **bC** clear all breakpoints

Syntax:

bc *addr*
bC

Description:

These commands delete breakpoints from the breakpoint table.

bc deletes the single breakpoint specified by the address *addr*, and **bC** deletes all breakpoints from the table.

- **bd** display breakpoints

Syntax:

bd

Description:

bd displays all entries in the breakpoint table.

For each breakpoint, the following information is displayed:

- Its address, using a symbolic name, if possible.
- The number of times reached without breakpointing.
- The skip count;
- The command list, if any.

For example, a **bd** display might be:

address	hits	skip	command
<code>_printf</code>	1	2	
<code>_putc</code>	0	0	<code>db _Cbuffs</code>

In this example, two breakpoints are in the table. The first is at the beginning of the function `_printf`; a breakpoint will be taken for it every third time it is reached, and no command will be executed. Given its current hit count, a breakpoint will be taken the second time `_printf` is reached.

The second breakpoint is at the function `_putc`; a breakpoint will be taken each time the function is reached, and will display memory, in bytes, starting at `_Cbuffs`.

- **bh** set memory hash breakpoint

Syntax:

bh *[range]*

Description:

bh performs a simple checksum on the specified range of memory. Breakpoints are automatically set at the entry and exit of all functions. The checksum is checked each time the debugger regains control of the task. If the checksum is ever detected to be different, the debugger will stop the program at that point.

- **bq** toggle low memory checksum

Syntax:

bq

Description:

When the debugger starts, it checksums the first 256 bytes of memory. Each time the debugger regains control, this checksum is calculated. If the

checksum differs from the saved value, a message is displayed and the program stopped. The new checksum is saved so multiple breakpoints are not generated.

The **bq** command toggles whether the checksum is done or not. Turning off checksumming will speed single-stepping since otherwise the checksum is computed for each instruction executed.

- **br** reset breakpoint counters

Syntax:

br [*addr*]

Description:

br resets the "hit" counter for the specified breakpoint which is at the address, *addr*. If *addr* is not given, the "hit" counters for all breakpoints in the breakpoint table are reset.

- **bs** set or modify a breakpoint

Syntax:

[#] **bs** *addr* [*cmdlist*]

Description:

bs enters a breakpoint into the breakpoint table, or modifies an existing entry.

The optional parameter **#** is the skip count for the breakpoint. If not specified, the skip count is set to 0, meaning that each time the breakpoint is reached it will be taken.

The optional parameter *cmdlist* is a list of debugger commands to be executed when the breakpoint is taken.

- **bt** toggle the trace mode flag
- **bT** toggle the return trace mode flag

Syntax:

bt
bT

Description:

bt and **bT** toggle the trace mode and return trace mode flags, respectively.

The state of the trace mode flag determines whether trace mode is enabled or disabled.

The state of the return trace mode flag determines whether the tracing of a function's return is enabled or disabled. If trace mode is disabled, the return trace mode flag has no effect.

Note that recursive calls will not handle the return trace mode correctly.

- **bu** user defined breakpoint

Syntax:

bu *addr*

Description:

bu sets the address of a user-defined breakpoint function.

This command sets a pointer to a function which is called each time the debugger regains control. The function has no effect when it returns a zero value. When a non-zero value is returned, a message displaying the value is displayed and the task stopped.

The function is called as a C function and is expected to follow the standard C register saving conventions. Passed as an argument to the function is the address of an array of longs which contain the tasks register values, includ-

ing the program counter and status register. The order of items in the array is:

```

registers d0-d7
registers a0-a7
status register
program counter

```

This function can be used to breakpoint on special conditions. For example, the program can be stopped when the **d0** and **d1** registers contain the same value. The function would look like this:

```

        moveq    #0, d0
        move.l   4(sp), a0
        move.l   (a0), d1
        cmp      4(a0), d1
        bne      1$,
        moveq    #1, d0
1$:     rts

```

Setting the address to zero disables the breakpoint.

The Clear Commands

► **cs** clear symbol table

Syntax:

cs

Description:

cs removes all symbols from the debugger's memory-resident symbol table.

The Display Commands

- **db** display memory in bytes
- **dw** display memory in words
- **dl** display memory as longs
- **d** display memory in last format

Syntax:

```
db [range]  
dw [range]  
dl [range]  
d [range]
```

Description:

The **db**, **dw** and **dl** commands display successive bytes, words and double words of memory, respectively. **d** displays memory using the last format specified; for example, if **d** is entered, and **db** was the last **DISPLAY MEMORY** command, then **d** will display bytes, too.

The starting address of the *range* parameter is optional; if not specified, it defaults to the ending address of the last display's *range*, plus one.

Each line of the display begins with the address, followed by a hexadecimal display of 16 bytes, 8 words or 4 double words, followed by an ASCII display, by bytes, of the same data. For the ASCII display, values falling outside the range 0x20 to 0x7f are displayed as a period.

If the ending address does not fall on a multiple of 16 bytes, only the number of bytes or words specified in the last line will be displayed.

- **dc** display all code symbols

Syntax:

```
dc
```

Description:

dc lists all the code symbols in the memory-resident symbol table and all user-defined symbols.

For each symbol, name and address are displayed.

- **dd** display all data symbols

Syntax:

dd

Description:

dd lists all the data symbols in the memory-resident symbol table.

For each symbol, name and address are displayed.

- **dg** display global values

Syntax:

dg

Description:

For each data symbol in the debugger's symbol table, **dg** displays the contents of the 16-bit field referenced by that symbol.

- **ds** display stack backtrace

Syntax:

ds

Description:

ds displays information about the current function, the function which called it, and so on, back to **_main()**, the Manx function which called your function **main()**.

For each function, the information consists of the function's name, the parameters passed to it, and the address to which it will return. The arguments are displayed as a series of 16-bit hex values. If an argument is actually of type long or double, it will be displayed as separate words.

ds determines the number of parameters by looking at the instructions which follow the address to which the function will return. **ds** assumes that the **a5** register points to the C stack frame for the current function, unless the current instruction is within 4 bytes of the start of the function.

The Go commands

- **g** execute the program
- **G** execute the program, without setting table breakpoints

Syntax:

```
[#]g [@ function] [addr] [cmdlist]
[#]G [@ function] [addr] [cmdlist]
```

Description:

The **g** commands transfer control of the processor to your program, at the address specified by **pc**. Your program then executes until it terminates, an error such as division by zero occurs, or a breakpoint is taken; control then returns to the debugger program.

The parameters to the **g** commands allow one or two temporary breakpoints to be set in memory before your program is executed.

The difference between the **g** and the **G** command is that the **G** command sets in memory just the breakpoints specified in the command itself, while the **g** command also sets the breakpoints specified in the breakpoint table.

The **#** and **addr** parameters define one of the temporary breakpoints that a Go command can set:

- **#** is the skip count for the breakpoint; it defaults to zero, meaning that the breakpoint is taken every time it is reached.
- **addr** is the address for the breakpoint;

The *@function* parameter specifies that a temporary breakpoint is to be set at the return address of the specified function. If the function is not specified, it defaults to the current function. If a function is specified, the breakpoint is set to the address to which the function will return. In this case, the breakpoint is not set until the function is entered; thus, in programs which call the function from several different places, the breakpoint will be set at the actual address to which the function will return.

The *; cmdlist* parameter defines a sequence of debugger commands, separated by semicolons, that the debugger is to execute once a breakpoint which is specified in the *go* command is taken. If this parameter is not specified, it defaults to the command list used for the last temporary breakpoint.

Before setting breakpoints and transferring control to your program, the debugger single-steps your program, (that is, causes it to execute one instruction). This allows you to transfer control to a location in the program at which there is a breakpoint, without immediately triggering a breakpoint and re-entry to the debugger.

The Load Commands

► *ls* load symbols

Syntax:

ls progfile

Description:

ls loads symbols from the specified program file into the debugger's memory-resident symbol table, after first clearing the memory-resident table of all but those symbols defined with the *v* command.

The Memory Modification Commands

► *ma* allocate some memory

Syntax:

ma val

Description:

ma allocates *val* bytes of memory and displays the address of the memory.

This command is mostly used to allocate memory for patches and possibly to hand assemble a small user break routine. The "." associated with memory change and display is set to the address as well making it possible to load patches or user functions from a file.

- **mb** modify bytes of memory
- **mw** modify words of memory
- **ml** modify double words of memory

Syntax:

```
mb  addr expr1[expr2 ...]
mw  addr expr1 [expr2 ...]
ml  addr expr1 [expr2 ...]
```

Description:

mb, **mw** and **ml** modify bytes and words of memory, respectively.

The parameter *addr* specifies the address of the first byte or word to be modified.

The *expr* parameters are expressions, whose resulting values are set in memory, with *expr1* set in the first byte or word specified, *expr2* set in the next higher byte or word, and so on.

The *expr* parameters can be separated by spaces or commas.

- **mc** compare memory

Syntax:

```
mc  range = addr
```


Description:

mc compares two blocks of memory and, for each comparison which fails, displays the corresponding address and value.

range specifies one of the blocks of memory. The second begins at *addr* and has the same length as the first block.

- **mf** fill memory

Syntax:

mf *range* = *expr*

Description:

mf sets each byte in a block of memory to a specified value.

The *range* parameter specifies the memory block, and *expr* an expression whose resulting value is the value to be set in the range.

- **mm** move memory

Syntax:

mm *range* = *addr*

Description:

mm copies one block of memory to another.

The *range* parameter specifies the source block and *addr* the starting address of the block to be modified.

- **ms** search memory

Syntax:

ms *range* = *expr1* [*expr2* ...]

Description:

ms searches a block of memory for a sequence of bytes having specified values. For each match, the corresponding address of the start of the string is displayed.

range specifies the block of memory. The *expr* parameters are expressions, each of whose resulting values is one byte of the search sequence.

The Radix Command

- **n** change radix

Syntax:

nx
n

Description:

This command changes the default radix (hexadecimal) for user input or debugger display. The suffix *x* is a single character **b**, **d**, **o**, or **x** which changes the default radix to binary, decimal, octal, or hexadecimal respectively. If suffix *x* is omitted, a message giving the current radix is displayed.

The radix can be forced to a given value by specifying one of the following prefixes before the desired number:

0x	hex
0o	octal
0b	binary

To display a number in decimal, the number must have a . (period) appended to it.

The "Print" Command

- **p** formatted print

Syntax:

p[format] [addr][,count]

Description:

p displays a formatted section of memory. Data items are converted to a displayable form as directed by the format conversion string *format*.

format is a list of format specifications, each of which defines the type of a data item and the conversion to be performed on it.

p works its way through the *format* string, converting and displaying data in memory as requested by the format string items. When **p** reaches an item in the *format* string, it converts the data at its current address as directed by the format item. When it finishes processing a format string item, it increments its current address by the size of the data item that it just processed, and processes the next data item as directed by the next format string item.

The *format* string is optional; if not specified, the *format* string used by the previous **p** command is used.

addr specifies the address of the first data item that **p** is to convert and display. If *addr* is not entered, the starting address is assumed to be the print command's current address. Normally, this is the address of the first byte beyond the last data item converted by the last **p** command. However, there is a *format* item that causes **p** to remember the address contained in the current data item, and then make that the current address after it finishes processing the entire format string.

count specifies the number of times that **p** is to work its way through the *format* string. Each time through, **p** begins at the current address that was left by the last time through. If *count* is not specified, it defaults to one time.

The format items have the form:

[rpt][indir_flg][size]desc_code

where

- *desc_code* is a single-letter code that defines the type of the data item and the conversion to be performed upon it. For example, the code **d** converts the two-byte binary value at the current address to decimal, and prints it. So if **var** is an **int**, the following would print its value in decimal:

pd var

The code **x** says “take the two-byte binary value at the current address, convert it to hexadecimal, and print the result”. So the hexadecimal value of **var** could be printed with the command:

px var

- *indir_flg*s is a string of zero or more ***** characters, which are indirection indicators specifying that the value at the current data item is a pointer to a chain of zero or more pointers, the last of which points to an object whose type and requested conversion are defined by *desc_code*.

To find the data object corresponding to a format item that has indirection indicators, **p** begins by setting its idea of the address of the data object to the current address. It then works its way from left to right through the indirection indicators; for each indicator it replaces its current idea of the data object address with the pointer that is in the field at this address. The data object address is distinct from the current address. At the end of this process, the **p** command's current address is simply incremented past the first pointer.

A ***** specifies that the pointer within the field referenced by the current data object address is four bytes long. This pointer is the offset component of the new data object address from the last segment referenced.

For example, if the variable **cp** is a pointer to a character string (that is, its declaration is **char *cp**), then the string pointed at by **cp** could be printed by the command

p*s cp

Here we have made use of the **s** *desc_code*, which specifies that the data object is a character string, and that the string's characters are to be printed, with possible modifications as noted below, up to a terminating null character. After this command, the **p** command's current address is set to the byte immediately following **cp**.

As another example, if **cpp** is a pointer to an array of pointers to character strings (that is, the declaration of **cpp** is `char **cpp`), then the string pointed at by the first element of the array could be displayed with the command

```
p**s cpp
```

Following this command, the **p** command's current address is set to the byte following **cpp**.

- The *rpt* parameter of a format item defines the number of times that the item is to be processed. It allows a sequence of *rpt* identical format items to be abbreviated by just one such item with a leading *rpt* count.

For example, if **a** is an array of **floats**, then the first five items in this array could be displayed with the command

```
p5f a
```

This command uses the fact that the *desc_code* to convert a four-byte floating point value at the current address to a displayable value is **f**. This command is equivalent to the command `pffffff a`. At the end of this command, the **p** command's current address is set to the address of the byte following the last displayed **float**.

- The *size* parameter of a format item defines the number of data items that are to be converted and printed. When the format item does not use indirection, *size* has the same effect as *rpt*; for example, in the `p5f a` command above, the 5 could be interpreted as being a *size* parameter instead of a *rpt* parameter.

When the format item uses indirection, the *size* parameter defines the number of data items to be converted and printed at the end of the indirection chain. For example, if a module defines **lpp** as a pointer to an array of pointers to an array of **longs** (the declaration of **lpp** is `long **lp`), then the following command will display the first four **longs** pointed at by the first element of the pointer array:

```
p**4D lpp
```

Here we have used the **D** *desc_code*, which specifies that a four-byte signed binary value is to be converted to decimal and printed. The following command would display the first three longs pointed at by the first three elements of the pointer array:

```
p3*4D *lpp
```

To demonstrate further the difference between the *rpt* and *size* fields in a format item, consider the format items **4*d** and ***4d**. The first causes the print command to take the item at the current address as a pointer, increment the current address by four, convert to decimal and print the two-byte value referenced by the pointer, and then repeat the process three more times. At the end of the process, the current address has been advanced by sixteen.

The second item causes the print command to again take the item at the current address as a pointer, increment the current address by four, and then convert to decimal and print the four successive two-byte values that begin at the address defined by the pointer. At the end of the process, the current address has been advanced by four.

As an example of the use of format strings containing several format items, consider the following code in a program that uses short data pointers:

```
struct {  
    int *ip;  
    float flt;  
    char *cp;  
} var = {&i, 3.14159, "ralph"};  
int i=2;
```

The command

```
p*d2-xf*s2-x var
```

will print

```
2xxxx 3.14159 ralph yyyy
```

where *xxxx* is the hexadecimal address of *i* and *yyyy* is the hexadecimal address of the string.

A complete list of *desc_codes*

- b** Convert to hexadecimal and print a byte.
- d** Convert to hexadecimal and print a 2-byte signed binary value.
- D** Convert to decimal and print a 4-byte signed binary value.
- f** Convert and print a four-byte float.
- F** Convert and print an eight-byte double.
- o** Convert an octal and print a two-byte field.
- O** Convert to octal and print a four-byte field.
- x** Convert to hexadecimal and print a two-byte field.
- X** Convert to hexadecimal and print a four-byte field.
- u** Convert to decimal and print an unsigned, two-byte value.
- U** Convert to decimal and print an unsigned four-byte value.
- p** Print a pointer in address form with translation.
- P** Print a pointer in address form without translation.
- c** Print a character with translation.
- C** Print a character without translation.
- s** Print a string up to a terminating null byte with translation.
- S** Print a string up to a terminating null byte without translation.

For the **C** and **S** *desc_codes* each character is printed as is, with no translations.

For the **c** and **s** codes, printable ASCII characters (that is, whose hex value is between 0x20 and 0x7f) are printed as is. A character whose hex value is less than 0x20 is printed as two characters: ^ followed by the printable character whose hex value equals the original character's value plus 0x40. A character whose hex value is 0x80 or greater is displayed as a ' character followed by the one or two characters that would be printed for the character whose hex value equals that of the original character less 0x80. For example, **0x41**, **0x1**, and **0x81** would be printed as **A**, **^A**, and **'^A**, respectively.

The following *desc_codes* can be used to assist in the formatting of the **p** output:

character	output
N or n	Output a newline character
R or r	Output a blank character
T or t	Output a tab character
"string"	Output "string"

These characters can be preceded by a count specifying number of characters or strings to be output.

The next group of *desc_codes* change the **p** command's notion of the current address. They do not cause any printing.

^	Back up the current address by the size of the last data item.
- or +	Back up or advance, respectively, the current address by size bytes, where size is a decimal value preceding the code. If size is not specified, it defaults to one byte.
A or a	Remember the pointer that is contained in the current data object: If this pointer is not null, set the p command's current address to this value after the entire format string has been processed.

If the pointer is null, set the **p** command's current address to the value it had before the entire format string was processed.

The **A** and **a** *desc_codes* are useful for printing the elements of a linked list. For example, consider the following code, which defines the structure for a symbol table item and declares **sym_head** to be a pointer to this structure. The program that uses this structure and field will chain symbol table items together and set a pointer to the head of the chain in **sym_head**.

```
struct symbol {  
    struct symbol * sym_next;  
    char *sym_name;  
    unsigned sym_val;  
} *sym_head;
```


The following command would display the symbol table item pointed at by **sym_head** and then set the **p** command's current address to the next symbol table item, which is pointed at by the **sym_next** field in the first item:

```
pA"symbol name="*snt"value="x sym_head
```

After this command is entered, you can display successive symbol table items by simply entering:

P

The **p** command's current address is correctly set to the next table item, and since a format string is not specified, the **p** command will use the one that it last used.

You can print out multiple symbol table items by entering a single **p** command. To do this, place a comma and the maximum number of items to be printed after the command's starting address. The command will follow the chain, printing symbol table items until it either prints the specified number of items or it prints an item whose **sym_next** pointer is null. In the latter case, it will terminate and leave the **p** command's current address set to the address of the last symbol table item. For example, entering:

```
pA"symbol name="*snt"value="x sym_head,100
```

will print symbol table items until it either prints 100 items or it prints an item having a null **sym_next** pointer.

The Quit command

➤ **q** quit the debugger

Syntax:

q

Description:

When more than one window is created using the **an** command, individual windows can be closed using the **q** command. When the last window is closed, the debugger terminates.

When a window is closed, if a task was stopped for debugging, the task is allowed to resume.

The Register command

- **r** register display

Syntax:

r

Description:

r displays and modifies the registers, including the status registers, of the program being debugged.

The parameter-less version displays the registers. If a 68881 is in use for this task, its registers are displayed as well, though they may not be modified.

The parameterized version modifies the contents of a register, with **<reg>** being the name of the register to be modified, and *expr* an expression whose resulting value is to be set into the register.

The Single Step commands

- **s** single step with display
- **S** single step without display
- **t** single step with display over subroutines
- **T** single step without display over subroutines

Syntax:

[#] s [*cmdlist*]

[#] S [*cmdlist*]

[#] t [*cmdlist*]

[#] T [*cmdlist*]

Description:

These commands **SINGLE STEP** your program; that is, they execute its instructions one by one. The **s** versions of the command single-step through each and every instruction. The **t** versions allow you to treat **jsr** and **bsr** instructions specially. The debugger does not gain control until the routine called by the instruction returns.

The optional **#** parameter specifies the number of instructions to be executed; it defaults to one instruction. However, a value of **0** specifies a continuous mode which will only stop by user intervention or when one of the other breakpoints is encountered.

The optional *cmdlist* parameter is a list of debugger commands to be executed after each single-step.

The commands differ in that **s** and **t** display information after each single-step, whereas **S** and **T** only display information after the last single-step.

The displayed information consists of the registers and a disassembly of the next instruction to be executed.

The Unassemble commands

- **u** unassemble memory, with symbols
- **U** unassemble memory, without symbols

Syntax:

u *range*
U *range*

Description:

These commands **DISASSEMBLE** a range of memory; that is, they display the assembly language instructions in the range.

The **u** and **U** commands differ in that the **u** command will make use of the symbol table during disassembly and the **U** command will not. Also, the **U** command displays, for each instruction, the hex value of each byte of the instruction, whereas the **u** command will not.

With the **u** command, the disassembly of an instruction which references memory displays the location as the symbol nearest to the location plus an offset, if possible. With the **U** command, the location is displayed as a hexadecimal value.

The *range* parameter specifies the area of memory to be disassembled. It gives the starting address, and either the number of instructions to be disassembled, or the ending address of the area.

The Variable commands

- **v** create a new symbol
- **V** modify the value of an existing symbol

Syntax:

```
v     symbol = addr  
V     symbol = addr
```

Description:

The **v** and **V** commands are used to create a new symbol or modify the value for an existing symbol, respectively, in the debugger's memory resident symbol table.

symbol is the name of the symbol being created or modified, and *addr* is its address.

The symbol will be classified as a code symbol.

The Macro command

- **x** macro command

Syntax:

```
xc  
xc = cmdlist  
x?
```

Description:

The **x** command defines or executes a sequence of debugger commands, called a **MACRO**. It can also list the defined macros.

A macro is associated with any letter, so up to 26 macros can be defined. Case is insignificant. A macro is defined by typing the letter **x**, followed by the letter with which the macro is to be associated. Then follows an **=** character and the macro's list of debugger commands, with the commands separated by semicolons. To execute a macro, type **x**, followed by the letter with which the macro is associated, followed by a carriage return.

Macros can be listed with the command **x?**. The output format is suitable for reading back in from a file.

The Display Expression Command

► **=** display the value of an expression

Syntax:

= *expr*

Description:

This command displays the value of an expression.

The expression is displayed in several formats: hexadecimal, signed decimal, unsigned decimal, octal, binary, and ASCII. If a symbol table has been loaded, the closest symbol is displayed as well.

The Redirect Input/Output Commands

► **<** take input from file
► **>** log output to file
► **>>** log commands only

Syntax:

<*file*
>*file*
>>*file*

Description:

These commands are used to temporarily redirect input and output to the files specified.

The < command causes the input which normally comes from the keyboard to be taken from the named file instead. This continues until the end of file is reached. This is especially useful for defining macros.

The > command causes the output of all commands to be displayed in the window and also written to the file specified. This continues until either a different file is specified using the > command or until the > command is used with no file name. In this case, the current file is closed and redirection stops.

The >> command causes a log of the commands only to be saved in a file. This is useful if a complicated set of commands is needed to get a program to a certain point for debugging. After the commands have been logged to the file, the file can be used as input via the < command to retrace the steps to reach the same point.

The Help command

➤ ? list commands

Syntax:

[*letter*]?

Description:

This command lists the debugger commands. For groups of related commands, the listing usually lists the first letter of the commands followed by a ?. You can get a listing of all the commands in such a group by typing the letter, the ?, and return. For example, the listing for the display commands is **d?**; thus you can type **d?**, followed by return to get a listing of all the display commands.

Command Summary

AMIGA COMMANDS

add	display device list
adi	display interrupt list
adl	display library list
adp	display port list
adr	display resource list
ai	display task information
ak	kill the current task
al/aL	wait for next task load with/without symbols
am	display memory information
an	make a new debug window
ap/aP	select task for post-mortem
aq	close all windows
ar	release current task
as/aS	select task to stop with/without symbols
at	display task list

BREAKPOINT COMMANDS

bb/bw/bl	set byte/word/long memory-change breakpoint
bc/bC	clear one/all breakpoints
bd	display the breakpoint table
bh	set checksum breakpoint
bq	toggle low memory checksum
br	reset the breakpoint counters
bs	set or modify a breakpoint
bt/bT	enable/disable trace mode
bu	set user defined breakpoint

CLEAR COMMANDS

cs clear all symbols

DISPLAY COMMANDS

db/dw/dl/d display memory in
bytes/words/longs/last format
dc/dd display code/data symbols
dg display global values
ds display stack backtrace

GO COMMANDS

g/G execute user's program

LOAD COMMANDS

ls load symbol

MEMORY MODIFICATION COMMANDS

ma allocate some memory
mb/mw/ml modify bytes/words/longs of memory
mc compare areas of memory
mf fill memory
mm move memory
ms search memory

RADIX COMMAND

n change the default radix for input and display

FORMATTED PRINT COMMANDS

p generate formatted print

QUIT COMMAND

q close current window

REGISTER COMMAND

r register display

SINGLE STEP COMMANDS

s/S single step with/without display

t/T single step with/without display
over calls

UNASSEMBLY COMMANDS

u/U assemble memory

VARIABLE COMMAND

v/V create/modify symbol

MACRO COMMAND

x define or modify a command macro

DISPLAY EXPRESSION

= display value of an expression

INPUT/OUTPUT COMMANDS

< take input from file

> log output to file

>> log commands to file

HELP COMMAND

? list debugger commands

Chapter 8 - Technical Information

This chapter discusses technical topics and topics that could not be conveniently discussed elsewhere.

It is divided into the following sections:

1. **Program Organization.** Discusses the factors that affect the memory organization of a program.
2. **Segmentation and Segmented Code.** Describes the partitioning of a program's executable code into several segments.
3. **Object Module Libraries.** Discusses the object module libraries provided with Aztec C.
4. **Assembly Language Functions.** Describes how to interface assembly language routines with C routines.
5. **Interrupt Handlers.** Describes routines to provide real-time response to events.
6. **Creating Subtasks.** Discusses programmer use of multi-tasking supported by Amiga.
7. **Floating Point Information.**

8. Building Programs that Run from the Workbench. Describes how to build programs that run from the Workbench.

9. Creating Detachable Programs.

Note: In this chapter, the number zero is represented by the character "Ø", in order to better distinguish between it and the letter "O".

Program Organization

Programs created with the Aztec C system are automatically split into several different sections by the compiler and linker. These sections are called segments. Code is split into these distinct segments so that the Amiga can most efficiently utilize its memory, because segments need not be contiguous, or next to each other, within the Amiga's memory space. The following segments are used for a program:

- one or more code segments
- an initialized data segment
- an uninitialized data segment
- a stack segment
- zero or more heap segments

The code segment(s) contain all of the program's executable code. You can specify how many code segments there will be, (up to 256) and what routines will be contained within each one. See below for more details.

The initialized data segment contains all of the program's initialized data. Initialized data is data which is initialized at the time of its declaration. For example, the following code fragment:

```
int i = 5;
main ()
{
...
}
```

declares **i** as an initialized data object. Note that only global variables and static variables which are initialized count as initialized data; normal local variables are always placed on the stack. Initialized data is set to a value at compile time, and thus takes up space in the executable file.

The uninitialized data segment is used for all uninitialized global and static variables. In the following code fragment, **i** is uninitialized data because it is given its initial value from within a function, and not where it is declared:

```
int i;
main ()
{
    i = 5;
}
```

Initialized data does not take up space in the executable file, but it must be initialized at runtime from within a function before it is used.

The stack segment is used for local (also known as automatic) variables, as well as for function call address information.

The heap segment(s) are used by the standard memory allocation routines (**malloc()**, **calloc()**, etc.), and indirectly by the standard I/O routines, to allocate memory at runtime. These segments do not take up any space within the program file.

PLACEMENT OF SEGMENTS OF MEMORY

The locations of the code, initialized and uninitialized data segments, and stack are set by the Amiga loader. The placement in memory of any particular program segment is independent of the placement of any other segments, with one exception: If any of the program's modules use the small data memory model, the program's initialized and uninitialized data segments occupy a single, contiguous block of memory, with the uninitialized data following the initialized data.

Linker options **+c** and **+f** can be used to force a program's code, initialized data, and/or uninitialized data segments to be loaded into chip or fast memory. For details, see the **Linker** chapter. A common application for this

is to force all initialized data into chip memory, so that the Amiga custom graphics chips can access this data.

SEGMENT SIZE

Due to various factors, certain segments may have size restrictions placed upon them. The exact restrictions depend on the segment type:

- **Code segments:** There is no limit on the size of a program's code segments, regardless of whether the small code or large code memory models are used.
- **Initialized and uninitialized data segments:** If all of a program's modules use the large data memory model, there is no limit to the sizes of its initialized and uninitialized data segments. If any of its modules uses the small data memory model, then register **a4** points 32K from the beginning of the program's data area, and data that small data modules attempt to access must lie within 32k bytes on either side of the location pointed at by **a4**. Because of this limitation, it is strongly recommended that you do not mix small and large data model files, unless you carefully plan how your files will be linked.
- **Stack segment:** When a program is loaded within the CLI environment, the size of its stack area is set to the value specified in the most recently-entered stack command, or to the CLI default value if a stack command has not been entered. When a program is loaded by the Workbench, the program's **.info** files define the size of its stack area.
- **Heap segment(s):** The size of a program's heap segments depends on its use of the dynamic allocation routines and standard I/O routines. If none of these routines are used, it is possible to effectively have no heap segment. For a discussion of the size of dynamically-allocated buffers which are placed in the heap segments, see the **Library Overview** chapter in your *Aztec C Library Manual*.

Segmentation and Segmented Code

The following two sections describe how the executable code portion of a program can be split into multiple segments to allow a large program to run within a limited amount of memory. The section entitled **SEGMENTATION** discusses how segmentation works, and the **SEGMENTED CODE** section describes how to create segmented code.

SEGMENTATION

Code segmentation is very similar to overlay systems which are available on other computer systems. The primary difference between the Amiga's segmentation scheme and other overlay techniques is that your source code does not need to be modified to take advantage of it. All segmentation setup is done at link time.

When segmentation is used, a program should be designed so that all major functions are mutually exclusive of each other. Thus, it is not necessary for each function to be available at all times. Instead, the control section of the program, called the root, loads each function only when needed. The root remains in memory at all times.

When a program is segmented, functions may exist within different segments. When the program is executed, only the root segment is initially loaded into memory. Then, when the root calls a function that is not located within a segment that has already been loaded, the segment that contains the function is automatically loaded into memory from disk. The segment remains in memory until the root segment explicitly removes it.

In reality, any reference from one segment to another segment causes the specified segment to be loaded, if it was not previously loaded in. Therefore, no restriction is imposed upon the structure of the program. In fact, one way to understand segmentation is to think of it simply as delayed program loading. If no segment is ever freed, it is possible to load all segments into memory at once if sufficient memory is present.

Loading a Segment

When a segment is loaded, the call to the segment loader tells it what segment is desired. The segment number is used to index into the segment

data table and get the file offset of the segment in the output file. The segment loader seeks to the offset and calls the DOS loader to load the segment. The hunks in the segment are added to the segment list of the program. "Hunk" is a subdivision of a segment.

Once the segment is loaded, the segment loader walks through the segment Jump Table, changing each entry into an absolute reference. It uses the hunk data tables to find the correct load address and the number of entries per hunk. See the *AmigaDos Technical Reference Manual* for more details on hunks and the Amiga executable file format.

Unloading a Segment

When a segment is unloaded, the above process occurs in reverse. The absolute jumps are converted back to the bsr and offset, the hunks of the segment are removed from the segment list, and the segment is unloaded.

Segments are only unloaded by manual calls from the program to unload a segment, using the **freeseq()** call. The argument is the address of a function in the segment.

For example, if **foo()** is declared to be a function in the segment to be unloaded, the call to **freeseq()** looks like:

```
int foo();  
freeseq(foo);
```

Segload

The segment loader itself is a routine that has the name **.segload**. When you specify segmentation, the linker searches for the **.segload** routine. A default version of **.segload** is supplied in the **c.lib** library as well as in the **lib** directory in the file **segload.o**.

Because **.segload** is not directly callable from a C routine, there is also a user callable routine, **segload()**, which takes as its argument the name of a function within the required segment. **segload()** then loads the appropriate segment into memory without actually calling a function in the segment. This can be used by a program that wants a number of segments to be loaded at the beginning of the program before any other activity occurs.

Linking and Startup Code

When segmented code is used, remember that the startup code for the program must always be in the root segment. If it is not, an error will occur in the linker. The startup code may be forced into the root segment in two ways—by either linking the `c.lib` library into the root segment, or linking just the individual module which contains the startup code into the root segment.

To place the user-referenced library functions into the root segment after all the other segments are linked, use

```
ln prog.o +o0 -lc
```

where `prog.o` represents your own object modules.

To put just the startup code into the root segment and not the entire library, link in either the `crt0.o` or `lcrt0.o` files (for either small code, small data or large code, large data, respectively) like this:

```
ln +o prog.o +o0 crt0.o +o -lc
```

(More about linking may be found under "Code Segmentation".)

Segment Construction

All references between segments are made through a segment Jump Table located in the data hunk. Initially, the reference contains the following 8 bytes:

<code>bsr</code>	<code>.segload</code>	<code>;segment loader routine</code>
<code>dc.b</code>	<code>segnum</code>	<code>;number of segment to load</code>
<code>dc.b</code>	<code>hioffset</code>	<code>;high 8 bits of 24 bit offset</code>
<code>dc.w</code>	<code>looffset</code>	<code>;low 16 bits of 24 bit offset</code>

After the segment is loaded, each entry looks like:

<code>jmp</code>	<code>absolute</code>	<code>;relocated address of symbol</code>
<code>dc.w</code>	<code>offset</code>	<code>;saved offset to .segload</code>

Segment Data Table

The segment data table has two parts. The first part consists of eight bytes each of segment data:

dc.l	seg_offset	;offset to segment in file
dc.w	tabl_offset	;offset of segment's entries
dc.w	hunk_offset	;relative offset of hunk data

Hunk Data

Following the segment data is the hunk data which consists of:

dc.w	hunk_num	;number of the hunk
dc.w	num_symbol	;number of symbols in Jmptab

The hunk data for each segment is terminated by a long word of zero.

SEGMENTED CODE

As stated, with Aztec C, you can create and execute programs that are larger than available memory by dividing a program's executable code into several segments. A segment is brought into memory when needed; when it is no longer needed, the memory it occupied can be released and reused.

This section describes the creation and use of segmented programs. It is divided into the following paragraphs:

- Programmer Information. Describes how you write programs having segmented code.
- Operator Information. Describes how you link programs having segmented code.

Programmer Information

There are two areas of concern for programs whose code is segmented: how segments are loaded and unloaded, and how programs access global and static data.

SEGMENTATION already described how to load and unload programs. Here we will deal with **Global and Static Data**.

Global and Static Data

There is only one global and static data segment for a program, regardless of the number of code segments it has. This segment is loaded automatically along with the program's root segment, and it remains in memory during the entire execution of the program, independent of the state of the program's code segments.

The name of each global variable in a program is unique: If several segments declare a global variable having the same name, they will both access the same variable.

Operator Information

All of a program's code segments are created during a single activation of the Linker.

The code for a command program can be divided into a maximum of 256 segments, each of which has an identifying number between 0 and 255. All command programs must have a code segment 0, which is the first segment loaded for the program.

The linker command that is used to create a command program whose code is segmented is identical to that which is used to create a command program whose code is unsegmented, except that the list of files are interspersed with the **+o** option. The **+o** option causes the object modules that follow it to be placed in a selected code segment.

A segment number can optionally be appended to option **+o**, to explicitly select the segment into which the following modules are placed. If a segment number is not specified for option **+o**, the modules are placed in the next available segment.

Example

The following command creates the command program **prog**, which has three code segments. Segment 0 contains the code for the modules **menu.o**, **subs.o**, and any needed modules from **c.lib**. Segment 1 contains the code for the modules **mod1.o** and **mod2.o**. Segment 2 contains the code

for `mod3.o` and `mod4.o`, and any `c.lib` modules referenced by segments 1 and 2 which are not in segment 0:

```
ln -f prog.lnk
```

where `prog.lnk` contains:

```
+o0 menu.o subs.o -lc
+o mod1.o mod2.o
+o mod3.o mod4.o -lc
```

All the files for this example could have been specified on the command line that activated the linker. We did not do this for two reasons: First, the entire command would not have fit on one line of this page. Second, you will also want to use `-f` files to link programs that have segmented code, since such programs usually have many modules, making it impractical to specify all the file and segmentation information on one line.

Including Modules from Libraries

This example illustrates a point about library searches during the linking of command programs having segmented code. Library code is always contained in the segment where the library is defined, not where it is referenced. For instance, if a module `x.o` in segment 2 references a function in `c.lib`, which is linked as `+o0 -lc`, then the referenced function will be placed in segment 0.

However, in the above example we do not wish to have an overly large segment 0, so the `c.lib` library is linked in twice. Therefore, when the `menu.o` and `subs.o` modules are linked, only the routines within the `c.lib` library that they reference will be contained within segment 0. When any of the other object modules reference functions with `c.lib`, those functions will be placed into segment 2.

Reselecting Segments

The Linker allows individual segments to be selected more than once on the link line. In this case, the modules specified following the reselection are appended to the code that is already in the segment. A segment can be reselected any number of times.

For example, the above example can be modified so that all **c.lib** modules referenced by all segments are included in segment $\emptyset$, by modifying **prog.link** as follows:

```
+o menu.o subs.o
+o mod1.o mod2.o
+o mod3.o mod4.o
+o $\emptyset$  -lc
```

The **+o $\emptyset$** reselects segment $\emptyset$, so that the modules pulled from **c.lib** are included in segment $\emptyset$.

Object Module Libraries

Several libraries of object modules are provided with Aztec C. The base libraries are:

- **c.lib** nonfloating point functions
- **mf.lib** Motorola Fast Floating Point (32-bit) math functions
- **ma.lib** Amiga IEEE floating point (64-bit)
- **m.lib** Manx IEEE floating point (96-bit)
- **m8.lib** 68881 math library
- **s.lib** screen functions

There are several versions of each library, depending on code, data, and integer size. Libraries which use 32 bit ints have the number 32 in their name; libraries which use large code/large data have the letter "l" in their name; libraries which make use of the 68881 coprocessor have an 8 in their name. For example, the large code/large data **c.lib** library is called **cl.lib**, and the large code/large data, 32 bit int **c.lib** library is called **cl32.lib**.

Assembly Language Functions

This section discusses assembly language functions that can be called by, and themselves call, C language functions. It first discusses the conventions that such functions must follow, and then discusses the in-line placement of assembly language statements within C language functions.

CONVENTIONS FOR C CALLABLE, ASSEMBLY LANGUAGE FUNCTIONS

A C callable, assembly language function must obey the conventions that are described in the following paragraphs.

Names of External Functions and Variables

The C compiler translates the name of a function or variable to assembly language by truncating the name to 31 characters and prepending an underscore character. Thus, assembly language modules that are to be accessible from C language modules or that are to access C modules must obey this convention.

For example, the following C language module calls the function `bmp()`, which simply adds 10 to the global short count. A C language module refers to this function as `bmp()`, and an assembly language module refers to it as `_bmp`.

```
int count;
main()
{
    bmp();
}
```

An assembly language version of `_bmp` could be:

```
#asm
    dseg
    public _count
    cseg
    public _bmp
_bmp:
    add.w    #10, _count
    rts
    end
#endasm
```

Note: `#asm` and `#endasm` are needed in order to incorporate assembly language routines into C code. See the section **Embedded Assembler Source** in this chapter for more information.

Function Calls and Returns

The assembly language code generated by the compiler for a C language call to another function pushes the arguments onto the stack, in the reverse order in which they were specified in the call's argument list, and then calls the function.

An assembly language function returns to a C function caller by issuing an `rts` instruction and leaving the caller's arguments on the stack. The caller then removes the arguments from the stack.

A function returns an integer, pointer, or Fast Floating Point value in register `d0`. IEEE Floating Point values are returned in the register pair `d0/d1`, unless using the 68881 which uses `fp0`.

For example, consider the following assembly language function, `_sub`, that takes two short arguments that are passed to it on the stack, subtracts them, and returns the difference as the function value. A C language function refers to this function using the name `sub`.

```
      cseg
      public      _sub
_sub:
      move.w      4(sp),d0      ;get first argument
      sub.w       6(sp),d0      ;subtract 2nd from first
      rts
```

The following C function calls `sub` to subtract `b` from `a`, and stores the difference in `c`

```
main()
{
    short a,b,c;
        c = sub(a,b);
}
```

Global Variables

A C module's global variables are in either the uninitialized data segment or in the initialized data segment.

An assembly language module can create an uninitialized variable that can be accessed by a C function, using the `global` directive. For example, the following code creates the global variable `_var`, which can be accessed as an array by a C function, and reserves 8 bytes of storage for it.

```
global _var, 8
```

A C function that wants to access `_var` could have the following declaration:

```
extern short var[];
```

An assembly language module can create an initialized variable that can be accessed by a C function using the `public` directive and the `dc` directive. For example, the following code creates the public variable `_ptr` that initially contains a pointer to the symbol `str`, and that can be accessed as a `char` pointer by a C function:

```
                dseg
                public  _ptr
_ptr            dc.l    str
```

To access `_ptr`, a C function could use the following declaration:

```
extern char *ptr;
```

An assembly language module can access global initialized or uninitialized variables that are created in C modules by defining the variables using a `public` directive that is in the `dseg` segment. For example, suppose a C

module creates a global, uninitialized **short** named **count** and a global, initialized **short** named **total** using the statement:

```
short count, total=1;
```

An assembly language module can access these variables by using the following directives:

```
dseg  
public      _count, _total
```

The above discussion assumes that the C modules and the assembly language modules follow the standard rule in the C language regarding external variables. This rule requires a global variable to be defined without the **extern** keyword in exactly one module and with that keyword in all other modules. Aztec C also supports a relaxed version of this rule. For information on this, see the **Compiler** chapter.

Register Usage

Register usage is implemented according to the Amiga guidelines. An assembly language function that is called by a C function must save and restore all registers it uses, except for **d0**, **d1**, **a0**, and **a1**. These are compiler temporary registers which may be used without being saved.

Embedded Assembler Source

Assembly language statements can be embedded in a C program between an **#asm** and an **#endasm** statement. The pound sign (#) must stand in column one of the line, and the letters must be lowercase.

Embedded assembler code must preserve the contents of all registers it uses, except for **d0**, **d1**, **a0**, and **a1**.

It should make no assumptions about the contents of the registers, since the code that the compiler currently generates for C statements may change in the future.

To be safe, a `#asm` statement should be preceded by a semicolon. This avoids problems in which the compiler mistakenly puts a label that is the target of a jump statement after, rather than before, in-line assembly code. Global variables may be referenced as described above.

Support is also provided to access predefined macros, function arguments, local variables, and file scope static identifiers. To access these from within a `#asm` block, precede the identifier name with `%%`. When passing output to the assembler, the compiler checks for the pattern `%%ident`.

If found, it checks to see if *ident* matches any predefined macros and replaces them with the actual constant values. It next checks for matches with any function arguments, local variables, or file scope static variables and replaces them with the appropriate address. If the identifier is not a macro, argument, auto, or static, the two percent characters and identifier are passed to the assembler output file.

In general, it is safest to contain assembly code in a separate, assembly language module rather than embedding it in C source.

Interrupt Service Routines

It is possible to write interrupt service routines (interrupt handlers) using Aztec C, without resorting to assembly language. Interrupt routines provide real-time response to events. Interrupts are serviced by the Amiga Exec system routines and, if an application program specifies, an interrupt service routine can be called in response to a particular interrupt.

When the interrupt occurs, the processor can be just about anywhere, with almost anything in its registers. The system interrupt handler preserves a number of registers and then calls the user service routine. However, it does not save all of the registers. In particular, it does not save registers **d2**, **d3**, and **a4**.

If the interrupt service routine modifies these registers, it must first preserve them. Since Aztec C considers **d2** and **d3** to be temporary registers, it does not preserve them. In addition, register **a4** is used as a base register to the global data of the program and should be modified if small code and data are being used.

There are two simple ways of preserving the necessary registers. First, if the program is compiled with option `-mcd`, the compiler preserves `d2` and `d3` at the entry to every function. In addition, this option forces the large code and data model. Therefore, register `a4` need not be modified and need not be saved. Note that the object modules need to be linked with the `cl` library.

The second way is necessary if you are not compiling with the `-mcd` option. Essentially, you should call two routines in the library at the beginning and end of your interrupt code which will save the necessary registers as well as setting up the `a4` register for small data references. The functions are called `—int_start()` and `int_end()`.

As an example, the interrupt handler `intfunc` looks like this:

```
intfunc()
{
    int_start();
    ...
    int_end();
}
```

The reason the `a4` register may sometimes need to be saved involves the different memory references models allowed by Aztec C. In the large model, all references are absolute and 32 bits. In the small model, references are 16 bits and either pc-relative or relative to some base register. For Aztec C, the base register is `a4`.

Therefore, all references are made relative to register `a4`. When the interrupt occurs, register `a4` might be pointing anywhere because almost any task might be executing, not only ours. If `a4` is not set up correctly by the interrupt service routine as the very first thing, the interrupt routine might use the `a4` register to access some global variables, which could create all sorts of problems.

Creating SubTasks

The Amiga's multi-tasking capability is very useful, since it is often easier for a separate task to handle asynchronous events without tying up the

main program. An excellent example of this is the **beep** program in the examples directory. A secondary task is created to actually make the beep sound whenever it gets a message from the parent task.

Creating the subtasks is discussed in the *Amiga Rom Kernel Manual* to some extent. However, there are a few special considerations when using Aztec C. In particular, when using small code and data, there is a little extra setup that must be done. In this case, the **a4** register is used as a base register to access global variables. Thus, for the program to operate correctly, **a4** must point at the proper memory location.

When a task is started, the registers contain somewhat arbitrary values. It is almost certain that **A4** will not contain the appropriate value. Therefore, the very first thing the task should do is set up the **a4** register. This is done by calling the routine **geta4()** as the first thing in the new task's function. For example:

```
main()
{
    void subtask();
    CreateTask("TASK", (long)PRI, subtask,
              (long)STKSIZ);
    ...
}

void subtask()
{
    geta4();
    ...
}
```

Floating Point Information

There are three floating point formats supported by Aztec C68k on the Amiga. Which one you choose depends on whether you are looking for speed, accuracy, or whether you have a math coprocessor on your machine.

HOW TO CREATE A FLOATING POINT FORMAT PROGRAM

When using floating point, there are two rules that **must** be followed for your program to work correctly.

- Compile with the desired floating point option
- Link with the corresponding floating point library and place that **before** the `c.lib`

FLOATING POINT FORMATS

The compiler generates code to use the appropriate format based on one of the following options: `-ff`, which is the Motorola Fast Floating Point; `-fa`, which is the Amiga IEEE Double Precision Floating Point Emulation; `-fm`, which is the Manx IEEE Double Precision Floating Point Emulation; and `-f8`, which is the 68881 Floating Point. These are all described in detail in the **Compiler** chapter.

The MathFFP library and the MathIeeeDoubBas library are opened on the first access of a floating point function when using the `mf.lib` or `ma.lib` library.

For example, if a program wished to use the Amiga IEEE format to achieve better floating point precision, the program would be compiled and linked this way:

```
cc -fa prog.c  
ln prog.o -lma -lc
```

Building Programs that Run from Workbench

OVERVIEW

This section discusses how to build programs that run from the Workbench.

First, however, read the *Amiga ROM Kernel Manual* which discusses the Workbench.

There are two ways for a program running from the Workbench to operate. First, the program can handle all its own I/O by creating windows, attaching a console device to the window, and getting input from the console and/or Intuition. In this case, there is not much to discuss because everything is handled by the programmer.

The second way is for the program to use the standard input and output channels that are assumed by the program to be preopened by the startup code. In this case, the program is doing most or all of its operations as though it were running on a simple line-oriented terminal. The trick here is to get a window opened by the startup code. This is accomplished by the `_wb_parse()` routine, which scans the Tool Array types, which are part of the Icon used to invoke the tool. It scans the array for a definition like the following:

```
WINDOW=xxxx
```

If found, the routine attempts to open the specified string and, if successful, sets `pr_CIS` and `pr_COS` to the window. Then, `_main()`, `stdin`, `stdout`, and `stderr` are set up for the window as well. If not found, `stdin`, `stdout`, and `stderr` are not initialized and any output is thrown away.

Note that this allows for debugging messages to be placed in a program and only displayed when the appropriate ToolType is set.

The only other note of importance is that when a program runs from the Workbench, the initial Workbench startup message is passed as `argv` to `main` with `argc` set to zero. However, for compatibility, the global variable `WBenchMsg` is initialized to point to the startup message as well.

STEP BY STEP

This section describes the step by step procedure to produce a program that displays the "Hello world!" message from the Workbench. Boot the

system using the **sys:1** disk and place the **sys:2** disk in the second drive. Type the following commands:

```
cd ram:
copy * hello.c

main()
{
    printf("Hello world!\n");
    getchar();
}
^ \
cc hello.c
ln hello.o -lc
```

At this point, run the hello program just to make sure it works from the CLI. The reason the **getchar()** routine is called is to give you time to see the message when you run from the Workbench.

Type:

```
iconed
```

The **IconEd** program starts and displays a screen with some information. Click the OK button and, in the menu, select **clear frame** to erase the existing picture. You may exercise your artistic freedom and make any kind of icon.

Once you have your icon designed, select **save** from the menus. A small requester appears. Change the filename to "hello". Click in the "Save entire image" box and close the window to exit the program. Type the following line:

```
loadwb
```

This command starts up the Workbench, so resize the CLI's window until you can see the disk icons. Click on the **Ram:** icon and you should see your

Hello icon. Click on this icon *once*. This selects it but does not attempt to execute it. Now, from the WORKBENCH menu, select INFO.

Selecting INFO should bring a window up with a number of gadgets and some information about the icon. Toward the bottom of the window is a single line: "TOOL TYPES string gadget". Click on the ADD button, click in the string gadget and type the following line:

```
WINDOW=CON:0/0/500/30/Hello
```

Then click on the ADD button again, and then click on the SAVE button in the bottom left-hand corner of the window.

That is all there is to it! Double click on the **Hello** icon now and the window should appear with the message waiting for a line to be typed. Press <CR> to end the program and the window disappears.

Chapter 9 - Error Messages

This chapter discusses error messages that can be generated by the compiler, assembler, and linker. It is divided into three sections: the first summarizes the compiler messages, the second the assembler, and the third discusses linker error messages.

A complete list of all error messages is provided on the next few pages. The error message explanations begin on page 9-11.

List of Error Messages

Compiler Error Messages

- 1: bad digit in octal constant
- 2: string space exhausted
- 3: unterminated string
- 4: argument type mismatch
- 5: invalid type for function
- 6: inappropriate arguments
- 7: bad declaration syntax
- 8: syntax error in typecast
- 9: invalid operand of & (address of)
- 10: array size must be positive integer
- 11: data type too complex
- 12: invalid pointer reference
- 13: unimplemented type
- 14: long switches not supported
- 15: storage class conflict
- 16: data type conflict
- 17: internal
- 18: data type conflict
- 19: bad syntax
- 20: structure redeclaration
- 21: missing }
- 22: syntax error in structure declaration
- 23: syntax error in enum declaration
- 24: need right parenthesis or comma in arg list
- 25: structure member name expected here
- 26: must be structure/union member
- 27: invalid typecast
- 28: incompatible structures
- 29: invalid use of structure
- 30: missing : in ? conditional expression
- 31: call of non-function

32: invalid pointer calculation
33: invalid type
34: undefined symbol
35: typedef not allowed here
36: no more expression space
37: invalid or missing expression
38: no auto. aggregate initialization allowed
39: enum redeclaration
40: internal [see error 17]
41: initializer not a constant
42: too many initializers
43: initialization of undefined structure
44: missing right paren in declaration
45: bad declaration syntax
46: missing closing brace
47: open failure on include file
48: invalid symbol name
49: multiply defined symbol
50: missing bracket
51: lvalue required
52: too many right paren's
53: multiply defined label
54: too many labels
55: missing quote
56: missing apostrophe
57: line too long
58: invalid # encountered
59: macro too long
60: loss of const/volatile info
61: reference to undefined structure
62: function body must be compound statement
63: undefined label
64: inappropriate arguments
65: invalid function argument
66: expected comma

- 67: invalid else
- 68: bad statement syntax
- 69: missing semicolon
- 70: goto needs a label
- 71: statement syntax error in do-while
- 72: statement syntax error in for
- 73: statement syntax error in for body
- 74: expression must be integer constant
- 75: missing colon on case
- 76: too many cases in switch
- 77: case outside of switch
- 78: missing colon on default
- 79: duplicate default
- 80: default outside of switch
- 81: break/continue error
- 82: invalid character
- 83: too many nested includes
- 84: constant expression expected
- 85: not an argument
- 86: null dimension in array
- 87: invalid character constant
- 88: not a structure
- 89: invalid use of register storage class
- 90: symbol redeclared
- 91: invalid use of floating point type
- 92: invalid type conversion
- 93: invalid expression type for switch
- 94: invalid identifier in macro definition
- 95: obsolete
- 96: missing argument to macro
- 97: too many arguments in macro definition
- 98: not enough arguments in macro reference
- 99: internal [see error 17]
- 100: internal [see error 17]
- 101: missing close parenthesis on macro reference

102: macro arguments too long
103: #else with no #if
104: #endif with no #if
105: #endasm with no #asm
106: #asm within #asm block
107: missing #endif
108: missing #endasm
109: #if value must be integer constant
110: invalid use of : operator
111: invalid use of a void expression
112: invalid use of function pointer
113: duplicate case in switch
114: macro redefined
115: keyword redefined
116: field width must be > 0
117: invalid 0 length field
118: field is too wide
119: field not allowed here
120: invalid type for field
121: ptr/int conversion
122: ptr & int not same size
123: far/huge ptr & ptr not same size
124: invalid ptr/ptr expression
125: too many subscripts or indirection on integer
126: too many arguments
127: too few arguments
128: #error
129: #elif with no #if
130: obsolete
131: ## at the beginning/end of macro body
132: obsolete
133: # not followed by a parameter
134: name of macro parameter was not unique
135: attempt to undefine a predefined macro
136: invalid #include directive

- 137: macro buffer overflowed
- 138: missing right paren
- 139: missing identifier
- 140: obsolete
- 141: invalid character
- 142: range-modifier ignored
- 143: range-modifier syntax error
- 144: invalid operand for sizeof
- 145: function called without prototype
- 146: constant value too large
- 147: invalid hexadecimal constant
- 148: invalid floating constant
- 149: invalid character on control line
- 150: unterminated comment
- 151: no block level extern initialization
- 152: missing identifier in parameter list
- 153: missing static function definition
- 154: function definition can't be via typedef
- 155: file must contain external definition
- 156: wide string literal not allowed here
- 157: incompatible function declarations
- 158: called function may not return incomplete type
- 159: syntax error in #pragma
- 160: auto variable not used in function
- 161: function defined without prototype
- 162: can't take address of register class
- 163: upper bits of hex character constant ignored
- 164: non-void type function must have return value
- 165: item not previously declared found in prototype

Fatal Compiler Error Messages

In the following "%%" is used to indicate a variable number of items can appear at that place.

out of memory!
unable to execute %%
too many -i options
illegal '-%%' option: %%
illegal '-h' option: '%%'
multiple '-h' options specified
more than one output file name specified
illegal option: '%%'
illegal 3.6 option: '%%'
illegal 3.6 '+x' option: '%%'
multiple input files specified!
can't open file '%%' for input!
no input file was specified!
cannot create output file '%%'!
too few arguments for '%%' option!
dump file not found!
pre-compiled header not in proper format!
pre-compiled header uses wrong size int!
error reading dump file!
error creating dump!!
Internal Errors

attempt to release item already free
attempt to use expression tree entry twice
optim failure
out of registers!
bad arginfo
bad op in cgen: 'op'
bad cast - 'cast'

Assembler Error Messages

Opcode Error Messages

addressing mode not valid

Need data or address register as index
Only W and L are valid modifiers
Scale must be 1, 2, 4, or 8
Field width must be in range (0-31)
Need data register here
Missing ':' in bit field syntax
Field width must be in range (1-32)
Missing '}' in bit field syntax
Need opcode, directive or macro name here
Unimplemented opcode
Size extension not allowed on this opcode
Post incrementing illegal in this mode
second argument must be immediate
Need register list as one of the operands
Need FP register list as one of the operands (mc68881)
An, Dn, modes not supported
illegal function code (mc68851)
pmove PSR, (ea) not allowed (mc68851)
Opcode extension size did not match
Bad size for expression
Opcode operands did not match
Illegal addressing format
Illegal argument expression
must be 16 bit
must be 8 bit
Need data register here
Missing argument
Bad argument
Invalid argument

Directive Error Messages

Illegal character in directive arguments
Equate directive requires a label
Invalid expression for equate directive

Need absolute value here
Set directive requires a label
Invalid expression for set directive
Name already defined
Multiply defined symbol
Register list directive requires a label
Need register list here
REG directive must contain register list
Macro end out of place
Need symbol for definition check
Requires two strings
Right side of test must match left side
Illegal character in conditional
Premature end of line in conditional string
Conditional end directive out of place
Need file name here
Unable to open include file
Need 'code' or 'data' here
Need second argument as well
Invalid expression for defined storage
Unimplemented assembler directive

Syntax Errors

bad '('
Missing ')'; Where's the ')'; Missing trailing
 parenthesis;
Missing ']'; illegal ']';
illegal '['
bad syntax
extra characters on line!
Unknown token in expression
Unknown opcode or directive
Illegal character in string
Bad character in line

Linker Error Messages

Command Line Errors:

Unknown option <bad option letter>
Too few arguments in command line
No input given!
Cannot have nested -F options.
Too few arguments in -F file: <filename>
Multiple entry points defined.
Invalid overlay number

I/O Errors

Cannot open <filename>, err = <errno>
Cannot open -F file: <filename>
I/O error (<errno>) reading/writing output file
Cannot write output file
Cannot create output file: <filename>
Cannot create symbol table output
Error creating .map file
Cannot create debugger symbol file<filename>
Error creating symbol listing file
Corrupted object files
Object file is bad!
Invalid operator in evaluate <hex value>
Library format is invalid!
Cannot read module from <input> on pass2
 or cannot find symbol, symbol name, on pass 2
Not an object file
Bad symbol typing information
Line displacement too large
Symbol name too long

Errors in Use of Memory

Insufficient memory!

Too many symbols!

Errors Arising From Source Code

Undefined symbol: <symbol name>

<symbol name> multiply defined

pass1(<hex value>) and pass2(<hex value>) values
differ:

or symbol type differs on pass two: <symbol name>

Branch out of range @pc=<addr>

Short branch to next location @pc=<addr>

Entry point must be in root segment

Entry point must be in first 64K of program

Attempt to store out of bounds

Program is too large to link

Attempt to perform relocation in overlay code

data ref to overlay code not in jump table

BigCrt0.o must be in root segment

Absolute reference from segment to segment

Excess number of segments

Can't do relocation from data to nonroot code

Data +BSS +JumpTab = 64K (code resource)

data reference out of range—remake with large data
model

Compiler Error Messages

Note:

- Error codes 116 and higher will not occur on Aztec C compilers whose version number is less than 3.
- Error codes greater than 200 will occur only if there is something wrong with the compiler. If you get such an error, please send us the program that generated the error.

1: **bad digit in octal constant**

The only numerals permitted in the base 8 (octal) counting system are zero through seven. In order to distinguish between octal, hexadecimal, and decimal constants, octal constants are preceded by a zero. Any number beginning with a zero must not contain a digit greater than seven. Octal constants look like this: 01, 027, 003. Hexadecimal constants begin with 0x (e.g., 0x1, 0xAA0, 0xFFF).

2: **string space exhausted**

The compiler maintains an internal table of the strings appearing in the source code. Since this table has a finite size, it may overflow during compilation and cause this error code. The table default size is about one or two thousand characters depending on the operating system. The size can be changed using the compiler option **-z**. Through simple guesswork, it is possible to arrive at a table size sufficient for compiling your program.

3: **unterminated string**

All strings must begin and end with double quotes ("). This message indicates that a double quote has remained unpaired.

4: **argument type mismatch**

This warning is given if the argument specified in a function call does not match that of the function's prototype. Although the warning is given, the

argument will be converted to the appropriate type before being passed. To avoid the warning, the argument can be preceded by a type cast to the appropriate type.

5: invalid type for function

Functions may be declared to return any scalar type as well as certain aggregate types such as structures. Functions are not allowed to return arrays. All definitions or declarations of a function or a function pointer that return an array will generate this error message. For example:

```
char(* f) () [];
```

6: inappropriate arguments

The declaration list for the formal parameters of a function stands immediately before the left brace of the function body, as shown below. Undeclared arguments default to `int`, though it is usually better practice to declare everything. Naturally, this declaration list may be empty, whether or not the function takes any arguments at all.

No other inappropriate symbols should appear before the left (open) brace.

```
badfunction(arg1, arg2)
short arg 1; /* misspelled or invalid keyword */
double arg 2;
{ /* function body */
}

goodfunction(arg1, arg2)
float arg1;
int arg2; /* this line is not required */
{ /* function body */
}
```

7: bad declaration syntax

A common cause of this error is the absence of a semicolon at the end of a declaration. The compiler expects a semicolon to follow a variable declara-

tion unless commas appear between variable names in multiple declarations.

```
int i, j;          /* correct */
char c d;          /* error 7 */
char *s1, *s2      /* error 7 detected here */
float k;
```

Sometimes the compiler may not detect the error until the next program line. A missing semicolon at the end of a `#include`'d file will be detected back in the file being compiled or in another `#include` file. This is a good example of why it is important to examine the context of the error rather than to rely solely on the information provided by the compiler error message(s).

8: syntax error in typecast

The syntax of the cast operator must be carefully observed. A common error is to omit a parenthesis:

```
i = 3 * (int number); /* incorrect usage */
i = 3 * ((int)number); /* correct usage */
```

9: invalid operand of & (address of)

This error is given if the program attempts to take the address of something that does not have an address associated with it.

```
#define FOUR 4
char *addr;

addr = &FOUR; /* error 9, can't take
               the address of a constant */
```

10: array size must be positive integer

The dimension of an array must be greater than zero. A dimension less than or equal to zero becomes 1 by default. As can be seen from the following example, a dimension of zero is not the same as leaving the brackets empty.

```
char badarray[0];           /* meaningless */
extern char goodarray[];    /* good */
```

Empty brackets are used when declaring an array that has been defined (given a size and storage in memory) somewhere else (that is, outside the current function or file). In the above example, **goodarray** is external. Function arguments should be declared with a null dimension:

```
func(s1,s2)
char s1[], s2[];
...
```

11: data type too complex

This message is best explained by example:

```
char *****foo;
```

The form of this declaration implies six pointers-to-pointers. The seventh asterisk indicates a pointer to a **char**. The compiler is unable to keep track of so many "levels". Removing just one of the asterisks will cure the error. However it is to be hoped that such a construct will never be needed.

12: invalid pointer reference

This error message will occur if pointer indirection is attempted on a type which cannot physically represent a pointer value. The only types, other than pointers themselves, which can hold pointer values, are **int**, **short**, and **long** (as well as their unsigned counterparts). All other C types will generate this error. For example,

```
char c;
*c = 5;
```

will generate this error.

13: unimplemented type

This error should not occur with the current compiler since all the ANSI specified data types are supported. In previous versions of the compiler, the **enum** keyword was allowed but the type was not supported.

14: long switches not supported

This error should not occur with versions 5.0 and higher, since long switches are supported.

15: storage class conflict

Only automatic variables and function parameters can be specified as **register**.

This error can be caused by declaring a **static register** variable. While structure members cannot be given a storage class at all, function arguments can be specified only as **register**.

A **register int i** declaration is not allowed outside a function--it will generate error 89 (see below).

16: data type conflict

The basic data types are not numerous, and there are not many ways to use them in declarations. The possibilities are listed below.

This error code indicates that two incompatible data types were used in conjunction with one another. For example, while it is valid to say **long int i**, and **unsigned int j**, it is meaningless to use **double int k** or **float char c**. In this respect, the compiler checks to make sure that **int**, **char**, **float** and **double** are used correctly.

<i>data type</i>	<i>interpretation</i>	<i>size(bytes)</i>
char	character	1
int	integer	2
unsigned/unsigned int	unsigned integer	2
short	integer	2
long/long integer	long integer	4
float	floating point number	4
long float/double	double precision float	8

17: internal

This error message should not occur. It is a check on the internal workings of the compiler and is not known to be caused by any particular piece of code. However, if this error code appears, please bring it to the attention of Manx, as it could be a bug in the compiler.

18: data type conflict

This message indicates an error in the use of the **long** or **unsigned** data type. **long** can be applied as a qualifier to **int** and **float**. **unsigned** can be used with **char**, **int** and **long**.

```

long i;                /* a long int */
long float d;          /* a double */
unsigned u;            /* an unsigned int */
unsigned char c;
unsigned long l;
unsigned float f;      /* error 18 */

```

19: bad syntax

This error occurs if the **#line** preprocessor directive is followed by something other than a numeric constant or macro that expands to one.

```

#line 100 "filename" /* correct */
#line "filename"     /* error 19 */

```

20: structure redeclaration

This message informs you that you have tried to redefine a **structure**.

21: missing }

The compiler requires a comma after each member in the list of fields for a structure initialization. After the last field, it expects a right (close) brace.

For example, this program fragment will generate error 21, since the initialization of the structure named **emily** does not have a closing brace:

```
struct john {  
    int bone;  
    char license[10];  
} emily = {  
    1,  
    "23-4-1984";
```

22: syntax error in structure declaration

This error occurs in a structure declaration that is missing the opening curly brace or when the left curly brace is followed by a right curly brace with nothing but white space.

```
struct    /* error 22, missing left curly brace */  
    int a;  
    long b;  
}
```

23: syntax error in enum declaration

This error occurs in an **enum** specification that is missing the opening curly brace or when the left curly brace is followed by a right curly brace with nothing but white space.

```
enum colors {  
}          /* error 23, nothing in enumerator list */
```

24: need right parenthesis or comma in arg list

The right parenthesis is missing from a function call. Every function call must have an argument list enclosed by parentheses even if the list is empty. A right parenthesis is required to terminate the argument list.

In the following example, the parentheses indicate that `getchar` is a function rather than a variable.

```
getchar();
```

This is the equivalent of

```
CALL getchar
```

which might be found in a more explicit programming language. In general, a function is recognized as a name followed by a left parenthesis.

With the exception of reserved words, any name can be made a function by the addition of parentheses. However, if a previously defined variable is used as a function name, a compilation error will result.

Moreover, a comma must separate each argument in the list. For example, error 24 will also result from this statement:

```
funcall(arg1, arg2 arg3);
```

25: structure member name expected here

The symbol name following the dot operator or the arrow must be valid. A valid name is a string of alphanumerics and underscores. It must begin with an alphabetic (a letter of the alphabet or an underscore). In the last line of the following example, `(salary)` is not valid because `'('` is not an alphanumeric.

```
emp_ptr = &anderson;
emp_ptr->salary = 12000;      /* these three lines */
(*emp_ptr).salary = 12000;   /* are */
anderson.salary = 12000;     /* equivalent */
emp_ptr = &anderson.;       /* error 25 */
emp_ptr- = 12000;           /* error 25 */
anderson.(salary) = 12000;   /* error 25 */
```

26: must be structure/union member

The defined structure or union has no member with the name specified. If the -s option was specified, no previously defined structure or union has such a member either.

Structure members cannot be created at will during a program. Like other variables, they must be fully defined in the appropriate declaration list. Unions provide for variably typed fields, but the full range of desired types must be anticipated in the union declaration.

27: invalid typecast

It is not possible to cast an expression to a function, a structure, or an array. This message may also appear if a syntax error occurs in the expression to be cast.

```
structure david { ... } amy;  
amy = (struct david)(expression );    /*error 27*/
```

28: incompatible structures

C permits the assignment of one structure to another. The compiler will ensure that the two structures are identical. Both structures must have the same structure tag. For example:

```
struct david emily;  
struct david amy;  
...  
emily = amy;
```

29: invalid use of structure

Not all operators can accept a structure as an operand. Also, structures cannot be passed as arguments. However, it is possible to take the address of a structure using the ampersand (&), to assign structures, and to reference a member of a structure using the dot operator.

30: missing : in ? conditional expression

The standard syntax for this operator is:

expression ? statement1 : statement2

It is not desirable to use ?: for extremely complicated expressions; its purpose lies in brevity and clarity.

31: call of non-function

Error 31 is generated by an expression that attempts to call a data item. The following code will generate an error 31:

```
int a;  
a();
```

Error 31 is often caused by an expression that is missing an operator. For example, Error 31 will be generated if the expression `a * (b + c)` is coded `a (b + c)`.

32: invalid pointer calculation

Pointers may be involved in three calculations. An integral value can be added to or subtracted from a pointer. Pointers to objects of the same type can be subtracted from one another and compared to one another. Since the comparison and subtraction of two pointers is dependent upon pointer size, both operands must be the same size.

33: invalid type

The unary minus (-) and bit complement (~) operators cannot be applied to structures, pointers, arrays and functions. There is no reasonable interpretation for the following:

```
int function();
char array[12];
struct joey , alice;
a = -array;
b = -alice;
c = ~function & WRONG;
```

34: undefined symbol

The compiler will recognize only reserved words and names which have been previously defined. This error is often the result of a typographical error or due to an omitted declaration.

35: typedef not allowed here

Symbols which have been defined as types are not allowed within expressions. The exception to this rule is the use of `sizeof(expression)` and the cast operator. Compare the accompanying examples:

```
struct lucille {
    int i;
}andrew;
typedef double bigfloat;
typedef struct lucille foo;
j =4 * bigfloat f;           /* error 35 */
k = &foo;                   /* error 35 */
x= sizeof(bigfloat);
y= sizeof(foo);             /* good */
```

The compiler will detect two errors in this code. In the first assignment, a typecast was probably intended; compare error 8. The second assignment makes reference to the address of a structure type. However, the structure type is just a template for instances of the structure (such as `andrew`). It is

no more meaningful to take the address of a structure type than any other data type, as in `&int`.

36: no more expression space

This message indicates that the expression table is not large enough for the compiler to process the source code. It is necessary to recompile the file using the `-e` option to increase the number of available entries in the expression table. For more information, see the **Compiler** chapter.

37: invalid or missing expression

This error occurs in the evaluation of an expression containing a unary operator. The operand either is not given or is itself an invalid expression.

Unary operators take just one operand; they work on just one variable or expression. If the operand is not simply missing, as in the example below, it fails to evaluate to anything its operator can accept. The unary operators are logical not (!), bit complement (~), increment (++), decrement (--), unary minus (-), typecast, pointer-to (*), address-of (&), and `sizeof`.

38: no auto. aggregate initialization allowed

Automatic arrays and structures may not be initialized. Static and external aggregates may be initialized, but by default their members are set to zero.

```
char array[5] = { 'a', 'b', 'c', 'd' };
function()
{
    static struct suzanne {
        int bone;
        char license[10];
    } daniel = {
        1,
        "123-4-1984"
    };

    char autoarray[2] = { 'f', 'g' };    /* no good */
    extern char array[];
}
```

There are three variables in the above example, two of which are correctly initialized. The variable **array** may be initialized because it is external. Its first four members will be given the characters as shown. The fifth member will be set to zero.

The structure **daniel** is static and may be initialized. Notice that **license** cannot be initialized without first giving a value to **bone**. There are no provisions in C for setting a value in the middle of an aggregate.

The variable **autoarray** is an automatic array. That is, it is local to a function and it is not declared to be static. Automatic variables reappear automatically every time a function is called, and they are guaranteed to contain garbage. Automatic aggregates cannot be initialized.

39: enum redeclaration

This error occurs when an **enum** identifier is used more than once in defining the value of enumeration constants.

```
enum states { NY, CA, PA }
enum states { IL, FL, NJ }
/* error 39, states has already been used */
```

40: internal [see error 17]**41: initializer not a constant**

In certain initializations, the expression to the right of the equal sign (=) must be a constant. Indeed, only automatic and register variables may be initialized to an expression. Such initializations are meant as a convenient shorthand to eliminate assignment statements. The initialization of statics and globals actually occurs at link-time, and not at run-time.

```
{
    int i = 3;
    static int j = (2 + i);           /* illegal */
}
```

42: too many initializers

There were more values found in an initialization than array or structure members exist to hold them. Either too many values were specified or there should have been more members declared in the aggregate definition.

In the initialization of a complex data structure, it is possible to enclose the initializer in a single set of braces and simply list the members, separated

by commas. If more than one set of braces is used, as in the case of a structure within a structure, the initializer must be entirely braced.

```
struct {
    struct {
        char array[];
    } substruct;
} superstruct =
version 1:
{
    "aBDdefghij"
};
version 2:
{
    {
        { 'a','b','c',...,'i','j' }
    }
};
```

In **version 1**, the initializers are copied byte-for-byte onto the structure, **superstruct**.

Another likely source of this error is in the initialization of arrays with strings, as in:

```
char array[10] = "aBDdefghij";
```

This will generate error 42 because the string constant on the right is null-terminated. The null terminator ('\0' or 0x00) brings the size of the initializer to 11 bytes, which overflows the ten-byte array.

43: initialization of undefined structure

An attempt has been made to assign values to a structure which has not yet been defined.

```
struct david {...};
struct dog david = { 1, 2, 3};      /* error 43 */
```

44: missing right paren in declaration

This error occurs in the declaration of a function pointer when the right parenthesis is left out.

```
int (* fp) ();          /* error 44 */  
int (* fp();           /* error 44 */
```

45: bad declaration syntax

This error code is an all purpose means for catching errors in declaration statements. It indicates that the compiler is unable to interpret a word in an external declaration list.

46: missing closing brace

All the braces did not pair up at the end of compilation. If all the preceding code is correct, this message indicates that the final closing brace to a function is missing. However, it can also result from a brace missing from an inner block.

Keep in mind that the compiler accepts or rejects code on the basis of syntax, so that an error is detected only when the rules of grammar are violated. This can be misleading. For example, the program below will generate error 46 at the end even though the human error probably occurred in the **while** loop several lines earlier.

As the code appears here, every statement after the left brace in line 6 belongs to the body of the **while** loop. The compilation error vanishes when a right brace is appended to the end of the program, but the results during run time will be indecipherable because the brace should be placed at the end of the loop.

It is usually best to match braces visually before running the compiler. A C-oriented text editor makes this task easier.

```
main()
{
    int i, j;
    char array[80];
    gets(array);
    i = 0;
    while (array[i]) {
        putchar(array[i]);
        i++;
        for ( i=0; array[i];i++) {
            for (j=i + 1; array[j]; j++) {
                printf("elements %d and %d are ",i, j);
                if (array[i] == array[j])
                    printf("the same\n");
                else
                    printf("different\n");
            }
        }
        putchar('\n');
    }
}
```

47: open failure on include file

When a file is **#included**, the compiler will look for it in a default area (see the **Compiler** chapter). This message will be generated if the file could not be opened. An open failure usually occurs when the included file does not exist where the compiler is searching for it. Note that a drive specification is allowed in an include statement, but this diminishes flexibility somewhat.

48: invalid symbol name

This message is produced by the preprocessor, which is that part of the compiler which handles lines which begin with a pound sign (#). The source for the error is on such a line. A legal name is a string whose first character is an alphabetic (a letter of the alphabet or an underscore). The

succeeding characters may be any combination of alphanumerics (alphabetic and numerals). The following symbols will produce this error code:

```
2nd_time,  
    dont_do_this!
```

49: multiply defined symbol

This message warns that a symbol has already been declared and that it is illegal to redeclare it. The following is a representative example:

```
int i, j, k, i;                /* illegal */
```

50: missing bracket

This error code is used to indicate the need for a parenthesis, bracket or brace in a variety of circumstances.

51: lvalue required

Only lvalues are allowed to stand on the left-hand side of an assignment. For example:

```
int num;  
num = 7;
```

They are distinguished from rvalues, which can never stand on the left of an assignment, by the fact that they refer to a unique location in memory where a value can be stored. An lvalue may be thought of as a bucket into which an rvalue can be dropped. Just as the contents of one bucket can be passed to another, so can an lvalue, *y*, be assigned to another lvalue, *x*:

```
#define NUMBER 512  
x = y;  
1024 = z;           /* wrong; rvalues are reversed */  
NUMBER = x;         /* wrong; NUMBER is an rvalue */
```

Some operators which require lvalues as operands are increment (++), decrement (--), and address-of (&). It is not possible to take the address of a register variable as was attempted in the following example:

```
register int i, j;  
i = 3;  
j = &i;
```

52: too many right paren's

This error should never be returned from the Amiga compiler.

53: multiply defined label

On occasions when the **goto** statement is used, it is important that the specified label be unique. There is no criterion by which the computer can choose between identical labels. If you have trouble finding the duplicate label, use your text editor to search for all occurrences of the string.

54: too many labels

The compiler maintains an internal table of labels which will support up to several dozen labels. although this table is fixed in size, it should satisfy the requirements of any reasonable C program. C was structured to discourage extravagance in the use of **gotos**. Strictly speaking, **goto** statements are not required by any procedure in C; they are primarily recommended as a quick and simple means of exiting from a nested structure.

This error indicates that you should significantly reduce the number of **gotos** in your program.

55: missing quote

The compiler found a mismatched double quote (") in a #define preprocessor command. Unlike brackets, quotes are not paired innermost to outermost, but sequentially. So the first quote is associated with the second, the third with the fourth, and so on. Single quotes (') and double quotes (") are entirely different characters and should not be confused. The latter are used to delimit string constants. A double quote can be included in a string by use of a backslash, as in this example:

```
"this is a string"  
"this is a string with an embedded quote: \". "
```

56: missing apostrophe

The compiler found a mismatched single quote or apostrophe (') in a #define preprocessor command. Single quotes are paired sequentially (see error 55). Although quotes can not be nested, a quote can be represented in a character constant with a backslash:

```
char c = '\';  
/* c is initialized to single quote */
```

57: line too long

Lines are restricted in length by the size of the buffer used to hold them. This restriction varies from system to system. However, logical lines can be infinitely long by continuing a line with a backslash-newline sequence. These characters will be ignored.

58: invalid # encountered

The pound sign (#) begins each command for the preprocessor: #include, #define, #if, #ifdef, #ifndef, #else, #endif, #asm, #endasm, #line and #undef. These symbols are strictly defined. The pound sign (#) must be in column one and lower case letters are required.

59: macro too long

Macros can be defined with a preprocessor command of the following form:

```
#define [identifier] [substitution text]
```

The compiler then proceeds to replace all instances of *identifier* with the *substitution text* that was specified by the **#define**.

This error code refers to the *substitution text* of a macro. Whereas ideally a macro definition may be extended for an arbitrary number of lines by ending each line with a backslash (\), for practical purposes the size of a macro has been limited to 255 characters.

60: loss of const/volatile info

This error occurs when passing the address of a variable that is declared as **const** and/or **volatile**.

```
extern const volatile int clock_time;  
set_time(&clock_time);           /* error 60 */
```

61: reference to undefined structure

This message comes in two forms:

- 1) As a warning, due to referencing an undefined structure member.
- 2) As an error, when trying to obtain the size of an undefined structure.

```
a = sizeof(struct nodef);  
/* error 61 unless nodef has been defined */
```

62: function body must be compound statement

The body of a function must be enclosed by braces, even though it may consist of only one statement:

```
function()  
{  
    return 1;  
}
```


This error can also be caused by an error inside a function declaration list, as in:

```
func(a, b)
int a; chr b;
{
    ...
}
```

63: undefined label

A **goto** statement is meaningless if the corresponding label does not appear somewhere in the code. The compiler disallows this since it must be able to specify a destination to the computer.

It is not possible to go to a label outside the present function (labels are local to the function in which they appear). Thus, if a label does not exist in the same procedure as its corresponding **goto**, this message will be generated.

64: inappropriate arguments

When a function is declared (as opposed to defined), it is poor syntax to specify an argument list:

```
function(string)
char *string;
{
    char *func1();           /* correct */
    double func2(x,y);      /* wrong */
    ...
}
```

In this example, **function()** is being defined, but **func1()** and **func2()** are being declared.

65: invalid function argument

This error occurs in a function definition that contains an argument that is not a valid identifier.

```
sub(a, 2b) { /* error 65 because identifiers
             can't begin with a numeric character */
```

66: expected comma

In an argument list, arguments must be separated by commas.

67: invalid else

An **else** was found which is not associated with an **if** statement. **else** is bound to the nearest **if** at its own level of nesting. So **if-else** pairings are determined by their relative placement in the code and their grouping by braces.

```
if(...) {
    ...
    if (...) {
        ...
    } else if (...)
        ...
    } else {
        ...
    }
```

The indentation of the source text should indicate the intended structure of the code. Note that the indentation of the **if** and **else-if** means only that the programmer wanted both conditionals to be nested at the same level, in particular one step down from the presiding **if** statement. But it is the placement of braces that determines this for the compiler. The example above is correct, but probably does not conform to the expectations revealed by the indentation of the **else** statement. As shown here, the **else** is paired with the first **if**, not the second.

68: bad statement syntax

The keywords used in declaring a variable, which specify storage class and data type, must not appear in an executable statement. In particular, all local declarations must appear at the beginning of a block, that is, directly following the left brace which delimits the body of a loop, conditional or function. Once the compiler has reached a non-declaration, a keyword such as `char` or `int` must not lead a statement; compare the use of the casting operator:

```
func()
{
    int i;
    char array[12];
    float k = 2.03;
    i = 0;
    int m;                /* error 68 */
    j = i + 5;
    i = (int) k;           /* correct */
    if (i) {
        int i = 3;
        j = i;
        printf("%d", i);
    }
    printf("%d%d\n", i, j);
}
```

This trivial function prints the values **3**, **2** and **3**. The variable `i` which is declared in the body of the conditional `if` lives only until the next right brace; then it dies, and the original `i` regains its identity.

69: missing semicolon

A semicolon is missing from the end of an executable statement. This error code is similar to error code 7. It will remain undetected until the following line and is often spuriously caused by a previous error.

70: goto needs a label

Compare your use of **goto** with this example. This message says that you did not specify where you wanted to **goto** with **label**:

```
    goto label;
    ...
label:
    ...
```

It is not possible to **goto** just any identifier in the source code; labels are special because they are followed by a colon.

71: statement syntax error in do-while

The body of a **do-while** may consist of one statement or several statements enclosed in braces. A **while** conditional is required after the body of the loop. This is true even if the loop is infinite, as it is required by the rules of syntax. After typing in a long body, do not forget the **while** conditional.

72: statement syntax error in for

This error occurs when the first of the two semicolons that separate the three expressions found in a **for** loop condition are missing.

```
    for (i=0 i; i++) {          /* error 72 due to
                                missing semicolon */
```

73: statement syntax error in for body

This error occurs when the second of the two semicolons that separate the three expressions found in a **for** loop condition is missing.

```
    for (i=0; i i++) {          /* error 73 due to
                                missing semicolon */
```

74: expression must be integer constant

This error occurs when a variable occurs instead of an integer constant in declaring the size of an array, initializing an element in an **enum** list, or specifying a **case** constant for a **switch**.

75: missing colon on case

This should be straightforward. If the compiler accepts a **case** value, a colon should follow it. A semi-colon must not be accidentally entered in its place.

76: too many cases in switch

The compiler reserves a limited number of spaces in an internal table for **case** statements. If a program requires more cases than the table initially allows, it becomes necessary to tell the compiler what the table value should be changed to. It is not necessary to know exactly how many are needed; an approximation is sufficient, depending on the requirements of the situation.

77: case outside of switch

The keyword, **case**, belongs to just one syntactic structure, the **switch**. If **case** appears outside the braces which contain a **switch** statement, this error is generated. Remember that all keywords are reserved, so that they cannot be used as variable names.

78: missing colon on default

This message indicates that a colon is missing after the keyword, **default**. Compare error 75.

79: duplicate default

The compiler has found more than one **default** in a **switch**. **switch** will compare a variable to a given list of values. But it is not always possible to anticipate the full range of values which the variable may take. Nor is it

feasible to specify a large number of cases in which the program is not particularly interested.

So C provides for a default case. The default will handle all those values not specified by a **case** statement. It is analogous to the **else** companion to the conditional, **if**. Just as there is one **else** for every **if**, only one default **case** is allowed in a **switch** statement. However, unlike the **else** statement, the position of a default is not crucial; a default can appear anywhere in a list of cases.

80: **default outside of switch**

The keyword, **default**, is used just like **case**. It must appear within the brackets which delimit the switch statement.

81: **break/continue error**

break and **continue** are used to skip the remainder of a loop in order to exit or repeat the loop. Break will also end a **switch** statement. But when the keywords, **break** or **continue**, are used outside of these contexts, this message results.

82: **invalid character**

Some characters simply do not make sense in a C program, such as '\$' and '@'. Others, for instance the pound sign (#), may be valid only in particular contexts.

83: **too many nested includes**

#includes can be nested, but this capacity is limited. The compiler will balk if required to descend more than three levels into a nest. In the example given, **file D** is not allowed to have a **#include** in the compilation of **file A**.

file A	file B	file C	file D
#include "B"	#include "C"	#include "D"	

84: constant expression expected

This error occurs when an integer constant is missing, such as in initializing an element in an `enum` list, specifying a `case` constant for a `switch`, or for a `#if` preprocessor directive.

85: not an argument

The compiler has found a name in the declaration list that was not in the argument list. Only the converse case is valid, i.e., an argument can be passed and not subsequently declared.

86: null dimension in array

In certain cases, the compiler knows how to treat multidimensional arrays whose left-most dimensions are not given in its declaration. Specifically, this is true for an `extern` declaration and an array initialization. The value of any dimension which is not the left-most must be given.

```
extern char array[][12];      /* correct */
extern char badarray[5][];    /* wrong */
```

87: invalid character constant

Character constants may consist of one or two characters enclosed in single quotes, as `'a'` or `'ab'`. There is no analog to a null string, so `''` (two single quotes with no intervening white space) is not allowed. Recall that the special backslash characters (`\b`, `\n`, `\t` etc.) are singular, so that the following are valid: `'\n'`, `'\na'`, `'a\n'`. `'aaa'` is invalid.

88: not a structure

Occurs only under compilation without the `-s` option. A name used as a structure does not refer to a structure, but to some other data type.

```
int i;
i.member = 3;                /* error 88 */
```

89: invalid use of register storage class

A globally defined variable cannot be specified as a register. Register variables are required to be local.

90: symbol redeclared

A function argument has been declared more than once.

91: invalid use of floating point type

Floating point numbers can be negated (unary minus), added, subtracted, multiplied, divided and compared; any other operator will produce this error message.

92: invalid type conversion

This error code indicates that a data type conversion, implicit in the code, is not allowed, as in the following piece of code:

```
int i;  
float j;  
char *ptr;  
i = j + ptr;
```

The diagram shows how variables are converted to different types in the evaluation of expressions. Initially, variables of type `char` and `short` become `int`, and `float` becomes `double`. Then all variables are promoted to the highest type present in the expression. The result of the expression will have this type also. Thus, an expression containing a `float` will evaluate to a `double`.

Types have the following hierarchy:

```
double float  
long  
unsigned  
int short, char
```


This error can also be caused by an attempt to return a structure, since the structure is being cast to the type of the function, as in:

```
int func()
{
    struct tag sam;
    return sam;
}
```

93: invalid expression type for switch

Only a char, int or unsigned variable can be switched. See the example for error 74.

94: invalid identifier in macro definition

This error occurs in a macro definition that contains one or more arguments that are not valid *identifiers*.

```
#define add(a,2b) (a+2b)    /* error 94 because
    identifiers can't begin with a numeric character */
```

95: obsolete

Error codes interpreted as obsolete do not occur in the current version of the compiler. Some simply no longer apply due to the increased adaptability of the compiler. Other error codes have been translated into full messages sent directly to the screen. If you are using an older version of the product and have need of these codes, please contact Manx for information.

96: missing argument to macro

Not enough arguments were found in an invocation of a macro. Specifically, a "double comma" will produce this error:

```
#define reverse(x,y,z) (z,y,x)
func(reverse(i,,k));
```

97: too many arguments in macro definition

This error occurs in a macro definition that contains more than 32 arguments in its definition.

98: not enough args in macro reference

The incorrect number of arguments was found in an invocation of a previously defined macro. As the examples show, this error is not identical to error 96.

```
#define exchange(x,y) (y,* error 98 */  
func(exchange(i));
```

99: internal [see error 17]**100: internal [see error 17]****101: missing close parenthesis on macro reference**

A right (closing) parenthesis is expected in a macro reference with arguments. In a sense, this is the complement of error 95; a macro argument list is checked for both a beginning and an ending.

102: macro arguments too long

The combined length of a macro's arguments is limited. This error can be resolved by simply shortening the arguments with which the macro is invoked.

103: #else with no #if

Correspondence between **#if** and **#else** is analogous to that which exists between the control flow statements, **if** and **else**. Obviously, much depends upon the relative placement of the statements in the code. However, **#if**

blocks must always be terminated by **#endif**, and the **#else** statement must be included in the block of the **#if** with which it is associated. For example:

```
#if ERROR 0
    printf("there was an error\n");
#else
    printf("no error this time\n");
#endif
```

#if statements can be nested, as below. The range of each **#if** is determined by a **#endif**. This also excludes **#else** from **#if** blocks to which it does not belong:

```
#ifdef JAN1
    printf("happy new year!\n");
    #if sick
        printf("i think i'll go home now\n");
    #else
        printf("i think i'll have another\n");
    #endif
#else
    printf("i wonder what day it is\n");
#endif
```

If the first **#endif** was missing, error 103 would result. And without the second **#endif**, the compiler would generate error 107.

104: **#endif** with no **#if**

#endif is paired with the nearest **#if**, **#ifdef** or **#ifndef** which precedes it. (See error 103.)

105: **#endasm** with no **#asm**

#endasm must appear after an associated **#asm**. These compiler-control lines are used to begin and end embedded assembly code. This error code indicates that the compiler has reached a **#endasm** without having found a previous **#asm**. If the **#asm** was simply missing, the error list should begin with the assembly code (which are undefined symbols to the compiler).

106: #asm within #asm block

There is no meaningful sense in which in-line assembly code can be nested, so the **#asm** keyword must not appear between a paired **#asm/#endasm**. When a piece of in-line assembly is augmented for temporary purposes, the old **#asm** and **#endasm** can be enclosed in comments as place-holders.

```
#asm
/* temporary asm code */
/* #asm      old beginning */
/* more asm code */
#endasm
```

107: missing #endif

A **#endif** is required for every **#if**, **#ifdef** and **#ifndef**, even if the entire source file is subject to a single conditional compilation. Try to assign pairs beginning with the first **#endif**. Backtrack to the previous **#if** and form the pair. Assign the next **#endif** with the nearest unpaired **#if**. When this process becomes greatly complicated, you might consider rethinking the logic of your program.

108: missing #endasm

In-line assembly code must be terminated by a **#endasm** in all cases. **#asm** must always be paired with a **#endasm**.

109: #if value must be integer constant

#if requires an integral constant expression. This allows both integer and character constants, the arithmetic operators, bitwise operators, the unary minus (-) and bit complement, and comparison tests.

Assuming all the macro constants (in capitals) are integers, then

```
#if DIFF = 'A' - 'a'
#if (WORD &= ~MASK) > 8
#if MAR | APR | MAY
```

are all legal expressions for use with **#if**.

110: invalid use of : operator

The colon operator occurs in two places: 1. following a question mark as part of a conditional, as in `(flag ? 1 : 0)`; 2. following a label inserted by the programmer or following one of the reserved labels, `case` and `default`.

111: invalid use of a void expression

This error can be caused by assigning a `void` expression to a variable, as in this example:

```
void func();  
int h;  
h = func(arg);
```

112: invalid use of function pointer

For example,

```
int (*funcptr) ();  
...  
funcptr++;
```

`funcptr` is a pointer to a function which returns an integer. Although it is like other pointers in that it contains the address of its object, it is not subject to the rules of pointer arithmetic. Otherwise, the offending statement in the example would be interpreted as adding to the pointer the size of the function, which is not a defined value.

113: duplicate case in switch

A **switch** statement has two **case** values which are the same. Either the two cases must be combined into one, or one must be discarded. For instance:

```
switch (c) {  
  case NOOP:  
    return (0);  
  case MULT:  
    return (x * y);  
  case DIV:  
    return (x / y);  
  case NOOP:  
  default:  
    return;  
}
```

The **case** of **NOOP** is duplicated, and will generate an error.

114: macro redefined

For example,

```
#define islow(n) (n>=0&& n<5)  
...  
#define islow(n) (n>=0&& n<=5)
```

The macro, **islow**, is being used to classify a numerical value. When a second definition of it is found, the compiler will compare the new substitution string with the previous one. If they are found to be different, the second definition will become current, and this error code will be produced.

In the example, the second definition differs from the first in a single character, '='. The second definition is also different from this one:

```
#define islow(n) n>0&& n>=<5
```

since the parentheses are missing.

The following lines will not generate this error:

```
#define NULL 0
...
#define NULL 0
```

But these are different from:

```
#define NULL '\0'
```

In practice, this error message does not affect the compilation of the source code. The most recent "revision" of the substitution string is used for the macro. But relying upon this fact may not be a wise habit.

115: keyword redefined

Keywords cannot be defined as macros, as in:

```
#define int foo
```

If you have a variable which may be either, for instance, a **short** or a **long** integer, there are alternative methods for switching between the two. If you want to compile the variable as either type of integer, consider the following:

```
#ifdef LONGINT
    long i;
#else
    short i;
# endif
```

Another possibility is through a **typedef**:

```
#ifdef LONGINT
    typedef long    VARTYPE;
#else
    typedef short   VARTYPE;
#endif
VARTYPE i;
```

116: field width must be > 0

A field in a bit field structure can not have a negative number of bits.

117: invalid 0 length field

A field in a bit field structure can not have zero bits.

118: field is too wide

A field in a bit field structure can not have more than 16 bits.

119: field not allowed here

A bit field definition can only be contained in a structure.

120: invalid type for field

The type of a bit field can only be of type `int` or `unsigned int`.

121: ptr/int conversion

The compiler issues this warning message if it must implicitly convert the type of an expression from pointer to `int` or `long`, or vice versa.

If the program explicitly casts a pointer to an `int` this message will not be issued. However, in this case, error 122 may occur.

For example, the following will generate warning 121:

```
char *cp;
int i;
...
i=cp;      /* implicit conversion of char to int */
```

When the compiler issues warning 121, it will generate correct code if the sizes of the two items are the same.

122: ptr & int not same size

If a program explicitly casts a pointer to an int, and the sizes of the two items differ, the compiler will issue this warning message. The code that is generated when the converted pointer is used in an expression will use only as much of the least significant part of the pointer as will fit in an int.

123: far/huge ptr & ptr not same size

This error occurs when trying to assign a near pointer to a far or huge pointer. A warning is generated when casting a far or huge pointer to a near pointer.

124: invalid ptr/ptr expression

If a program attempts to assign one pointer to another without explicitly casting the two pointers to be of the same type, and the types of the two pointers are in fact different, the compiler will issue this warning message.

The compiler will generate code for the assignment, and if the sizes of the two pointers are the same, the code will be correct. But if the sizes differ, the code may not be correct.

125: too many subscripts or indirection on integer

This warning message is issued if a program attempts to use an integer as a pointer; that is, as the operand of a star operator.

If the sizes of a pointer and an int are the same, the generated code will access the correct memory location, but if they are not, it will not.

For example,

```
char c;  
long g;  
*0x5c=0; /* warning 125, because 0x5c is an int */  
c[i]=0; /* warning 125, because c+i is an int */  
g[i]=0; /* error 12, because g+i is along */
```

126: too many arguments

This error occurs when a function is invoked with more arguments than is specified in its prototype or definition. The only exception allowed is when a variable number of arguments is specified in the prototype.

127: too few arguments

This error occurs when a function is invoked with less arguments than is specified in its prototype or definition.

128: #error

This error is generated by the **#error** directive and is followed by the optional sequence of preprocessing tokens found in the source code.

129: #elif with no #if

This error occurs when the **#elif** preprocessor directive is used without a preceding **#if** directive.

130: obsolete [see error 95]**131: ## at the beginning/end of macro body**

This error occurs when a **##** is found as the first or last end of a macro definition body.

```
#define TWOSHARP 2##          /* error 131 */
```

132: obsolete[see error 95]**133: # not followed by a parameter**

This error may occur in a **#define** macro in which the **#** operator is applied to a parameter in the replacement list. If the **#** token is not followed by a parameter, this error message will be generated.

134: name of macro parameter was not unique

This error should never be returned from the Amiga compiler.

135: attempt to undefine a predefined macro

This error occurs when attempting to use a `#undef` on those macros that are predefined by the compiler, such as `__STDC__`, `__TIME__`, `__DATE__`, `__FILE__`, `__LINE__`, and `__FUNC__`.

```
#undef __TIME__          /* error 135 */
```

136: invalid #include directive

This error occurs when the `#include` directive is not followed by a string literal or a *filename* enclosed in `< >` signs.

```
#include filename        /* error 136 */
```

137: macro buffer overflowed

This error should never be returned from the Amiga compiler.

138: missing right paren

This error occurs when attempting to use the defined directive with a left parenthesis and no matching right parenthesis.

```
#if defined(AMIGA        /* error 138 */
```

139: missing identifier

This error occurs when attempting to use the defined directive with no identifier following the defined keyword.

```
#if defined                /* error 139 */
```

140: obsolete

[see error 95]

141: invalid character

This error should never be returned from the Amiga compiler.

142: range-modifier ignored

Using a range modifier (**near**, **far**, etc.) on a structure or union member is allowed by the parser but has no effect and is ignored.

Note: This error should never be returned from the Amiga compiler.

143: range-modifier syntax error

A range-modifier is illegal as part of a function declaration. You cannot say that a function is **near**, **far**, **huge**, etc.

Note: This error should never be returned from the Amiga compiler.

144: invalid operand for sizeof

This error occurs when attempting to obtain the **sizeof** of something other than a previously defined data structure.

```
sub()  
{  
  int i = sizeof(sub); /* error 144 */  
}
```

145: function called without prototype

This warning is generated for functions that are called without having been prototyped. It only occurs when compiling with the **-wp** option.

146: constant value too large

This error occurs when attempting to use a constant larger than the unsigned long 0xffffffff in an expression.

147: invalid hexadecimal constant

This error occurs when the character following a 0x or 0X is not a valid hexadecimal constant.

```
int i = 0xg2;      /* error 147, 'g' is not  
                  a valid hex constant */
```

148: invalid floating constant

This error occurs if the first letter excluding the optional sign following the e or E in a floating point number is something other than a digit.

```
double d = 123e+f;  
          /* error 148, 'f' is not a digit */
```

149: invalid character on control line

This error occurs on conditional preprocessor lines that expect a single constant expression but get extra information.

```
#if CONST invalid  /* error 149 due to  
                  extra characters "invalid" */
```

150: unterminated comment

This error occurs if the start of a comment (/*) is not terminated with (*/) before the end of the file.

151: no block level extern initialization

This error occurs when initialization of an extern variable is attempted inside a function. Initialization of externs is permissible outside of func-

tions, or the function can declare the variable as **extern** and then initialize it further down in the code using an assignment statement.

152: missing identifier in parameter list

This error occurs when the type of an argument is specified in a function definition without being followed by the argument itself.

```
sub(int ) {           /* error 152, missing the
                        name of the int argument */
```

153: missing static function definition

This error occurs if a function has been declared as static in a file and has not been followed by its actual definition further down in the file.

154: function definition can't be via typedef

This error occurs when incorrectly defining a function using a **typedef**. It is possible to define a **typedef** that is a function such as:

```
typedef int F(void);
```

which sets the type **F** to be a function with no arguments returning **int**. Then, a function can be declared such as:

```
F f;
```

which is legal. However, the function definition:

```
F f {}
```

is illegal.

155: file must contain external definition

This error occurs in a file with no external data or function definitions when compiling with the **-pa** option to use the ANSI preprocessor.

156: wide string literal not allowed here

This error occurs when attempting to use a wide string literal with the `#include` or `#line` directives.

```
#include L"filename"          /* error 156 */
```

157: incompatible function declarations

This error occurs if a function declaration does not match a previous definition or declaration for the same function.

158: called function may not return incomplete type

This error occurs when a function attempts to return a structure which has not been defined. If a function is called that returns a structure, but the size of the structure is unknown, then it is not possible for the compiler to know how much data is being returned by the function and how much space to reserve for the return value.

For example:

```
struct foo x();

main()
{
    x();
}
```

159: syntax error in #pragma

This error occurs if the `#pragma` is used for a function call and does not match the following syntax.

```
#pragma regcall([return=]
    func(arg1,arg2,...,argn))
#pragma amicall(base,
    offset,func(arg1,arg2,...,argn))
#pragma libcall func base offset regmask
#pragma syscall func offset regmask
```

160: auto variable not used in function

This warning occurs when compiling with the **-wu** option and a function containing a local variable has not been used.

161: function defined without prototype

This warning occurs:

- when compiling with the **-wp** option and a function does not have its arguments prototyped

or

- if there are no arguments but you have not specified **void**.

162: can't take address of register class

This error occurs when attempting to take the address of a variable that has been declared as a register class variable.

```
register int a;  
int *ip;  
  
cp = &a;                               /* error 162 */
```

163: upper bits of hex character constant ignored

This warning occurs if the compiler encounters a hexadecimal character constant (specified by **\x**) whose value cannot fit within a single byte. For example:

```
char *cptr = "\x9b7";
```

will generate a warning because **"\x9b7"** cannot be stored within one byte. The compiler will ignore the most significant bits, and use the least significant bits. In the example the compiler will treat **"\x9b7"** as **"\xb7"**. This warning will occur if you accidentally place a digit from 0 through 9 or a letter from a through f immediately after a **\x** escape sequence. In the

example if you intended to have 0x9b followed by the ASCII digit 7, you could use string concatenation to produce the desired result:

```
char *cptr = "\x9b" "7";
```

164: non-void type function must have return value

This error message can occur only if the `-wr` compiler option is used. If the compiler encounters a function which is defined as returning a value (`int`, `char`, or `t` he like) but which does not have an explicit return. For example:

```
int func()
{
    printf("hello \n");
}
```

would generate this message. Replacing `int func()` with `func()` will not correct the error. The specification `void func()` will correct the problem. The specification of an explicit `return` will also.

165: item not previously declared found in prototype

This warning message will be generated if a `struct` appears as an argument in a function prototype and there is no previous declaration for the `struct`. This warning will be generated if there is no `struct` declaration or if the `struct` declaration occurs after the prototype statement. The sequence `struct` declaration, prototype statement, function definition will correct the problem as in this example:

```
struct astruct {
    int a;
    char c;
}

void func(struct astruct arg);

void func(struct astruct arg);
{
    ...
}
```

This problem presents special difficulties when it arises because the intended **struct** decalartation occurs after the prototype definition. A strict interpretation of ANSI rules, in this case, produces results that may seem arbitrary and illogical. Positioning the **struct** declaration so that it occurs before the prototype definition corrects the problem. If the problem is not corrected, it is unlikely that the program will run correctly.

Fatal Compiler Error Messages

If the compiler encounters a "fatal" error, one which makes further operation impossible, it will send a message to the screen and end the compilation immediately.

out of disk space!

There is no room on the disk for the output file of the compiler. Previous disk files will not be overwritten by the compiler's assembly language output. To make room on the disk, it is usually sufficient to remove unneeded files from the disk.

illegal '-%%' option:

The compiler has been invoked with an option letter which it does not recognize. The manual explicitly states which options the compiler will accept. The compiler will specify the invalid option letter.

multiple '-h' options specified

Only one **-hi** or **-ho** option can be specified.

more than one output filename was specified

Output from the compiler can only be directed to one file. More than one **-o** option was found.

duplicate output file

If an output file name has been specified with the **-o** option and that file already exists on the disk, the compiler will not overwrite it. **-o** must specify a new file.

too few arguments for -o option

The compiler expected to find the output filename following the **-o**, but did not find it. The output file name must follow the option letter, and the name of the file to be compiled must occur last in the command line.

can't open file '%%' for input

The input file specified in the command line does not exist on the disk or cannot be opened. A path or drive specification can be included with a filename according to the operating system in use.

no input file was specified!

While the compiler was able to open the input file given in the command line, that file was found to be empty.

multiple input files specified

More than one input file was specified

cannot create output file '%%'!

The compiler was unable to create an output file. On some systems, this error could occur if a disk's directory is full. It may also occur if the disk is locked.

out of memory!

Since the compiler must maintain various tables in memory as well as manipulate source code, it may run out of memory during operation. The

only solutions are to add more memory to your computer or to divide the source file into modules which can be compiled separately. These modules can then be linked together to produce a single executable file.

no disk space!

An error occurred in closing the assembly output file being written by the compiler. The compile therefore failed and the assembly file was deleted. Make additional room on the current disk or the one to which the **CCTEMP** environment variable is pointing.

unable to execute as

An attempt to launch the assembler failed. Check to make sure the assembler has not been renamed and that it can be located in your current execution path.

too many -I options

The number of include file paths has exceeded the maximum of 16 allowed. Try rearranging the include files into fewer directories to avoid the need to specify so many include file paths.

illegal "%%" option: "%%"

A secondary option was specified that is not recognized in conjunction with the first letter given. The error message gives the primary and illegal secondary option used. See the **Compiler** chapter of the manual, which explicitly states the options that the compiler will accept.

illegal 3.6 option: '%%'

The version 5.0 compiler will accept 3.6a options if the **-3** option was specified. Consult a 3.6 manual for specific option details. also see discussion of **-3** and **-5** options.

duplicate symbol dump file

Indicates that the **-h** option has already been used with a dump file name. The compiler only accepts one dump file as input and cannot generate an output dump file if one has already been specified for input. Use the **#include** directive multiple times in a single file to include all the source files that are to be precompiled, and then precompile that single file.

unable to execute %%

The compiler can chain to other programs such as the assembler or editor. There are various environment variables and option specification that determine the programs the compiler should launch and where it should look for them. Either the program does not exist on the disk, or the compiler was misled as to its whereabouts.

too few arguments for -%% option

The option expects a filename to follow.

dump file not found!

This error indicates a failure on attempting to open the specified dump file used with the **-h** option. Check to make sure the correct name was used on the command line following the **-h** option.

pre-compiled header not in proper format!

This error indicates that a precompiled dump file opened for reading did not contain the correct information to indicate that it is a valid dump file. The file is either not a dump file or the dump file has become corrupted and should be rebuilt.

error reading dump file!

This error indicates that a precompiled dump file that was opened for reading specified a length field that was longer than that actual dump file. The dump file has been corrupted and should be rebuilt.

error creating dump

This indicates that an error occurred in trying to open the dump file specified following the **-ho** option. Check to make sure the disk is not locked or full.

pre-compiled header uses the wrong size int

The compiler will treat ints as 16 bits or 32 bits according to option settings. When a pre-compiled header file is used, it is mandatory that equivalent data size options be in effect.

Internal Errors

The following fatal compiler error messages are internal. None of these errors should normally occur. If you get one, it may point to an internal compiler problem. Please contact Technical Support to report the problem; include some sample code to demonstrate it.

```
attempt to use expression tree entry twice
attempt to release item already free
optim failure.
out of registers!
bad op in cgen: 'op'
bad cast - 'cast'
```

Assembler Error Messages

This section describes the error messages that the Assembler generates as it is creating an object module.

Opcode Error Messages**68020 addressing mode not valid**

To enable assembly of 68020 addressing modes insert the directive:

```
machine mc68020
```

Need data or address register as index
Only W and L are valid modifiers
Scale must be 1, 2, 4, or 8

The format of the index operand for 68020 register indirect with index modes is "Xn.SIZE*SCALE". (See your 68020 reference manual.)

Field width must be in range (0-31)
Need data register here
Missing ':' in bit field syntax
Field width must be in range (1-32)
Missing '}' in bit field syntax

Bit fields are used in the following 68020 instructions:

bfchg, bfcclr, bfxexts, bfxextu, bfffo, bfinv, bfbset, bfbtst

These instructions operate on a string of consecutive bits in a bit array.

The syntax for a bit field is:

{offset.width}

The braces and colon must be included as shown. *offset* and *width* parameters must either be data registers or absolute expressions. *offset* must be in range (0-31). *width* must be in range (1-32).

Need opcode, directive or macro name here

Label names must be defined starting in the first column of the line.

Unimplemented opcode

Assembler does not understand opcode. Check your syntax.

(Note: The mnemonic TRAPcc has been changed to Tcc in the parameterless form, and TPcc is used when immediate data is specified. Similar changes have been made for FTRAPcc (mc68881) and PTRAPcc (mc68851).)

Size extension not allowed on this opcode

Many 680X0 opcodes have only one size or are specified 'unsized'.

Example:

```
pea.w (a0) ;      ERROR
pea (a0) ;        OK
```

Post incrementing illegal in this mode

Post incrementing (An)+ is only legal with certain opcodes.

Example:

```
add a1+,d0 ;      ERROR
add (a1)+,d0 ;    OK
```

second argument must be immediate

Should only occur when using pflush (mc68851). The mask must be immediate.

Need register list as one of the operands

A register list is used with movem.

Examples:

```
d3-d7          means d3, d4, d5, d6, and d7
d5/a5          means d5 and a5
d2-d4/a0-a3    means d2, d3, d4, a0, a1, a2, and a3
```

Need FP register list as one of the operands (mc68881)

fp register lists are used with fmovem. The syntax for a fp register list is any combination of fpcr, fpsr, and fpia, with individual register names separated by a slash "/".

Examples:

```
fpcr/fpsr  
fpcr/fpiar/fpsr
```

An, Dn, modes not supported

Certain 68851 opcodes will not accept *an* or *dn*. (See your 68851 reference manual).

illegal function code (mc68851)

Function codes must be expressed in any of the following registers:

dn, *sfc*, *dfc*, or as an immediate four bit number.

Examples:

```
ploadr d4, (a5)  
ploadr sfc, (a5)
```

pmove PSR, (ea) not allowed (mc68851)

(consult your 68851 reference manual)

Opcode extension size did not match

Bad size for expression

Occurs when you try to use an illegal size extension.

Opcode operands did not match

Illegal addressing format

Illegal argument expression

The address mode you are trying to use is not supported by the opcode.

must be 16 bit

Data must be 16 bit (\$0000 - \$ffff)

must be 8 bit

Data must be 8 bit (\$00 - \$ff)

Need data register here

Many opcodes need a data register (d0,d1,d2...d7) in the operand field. You can usually substitute **adda** for **add**, **suba** for **sub** and **cmpa** for **cmp**. (Consult your 680X0 reference manual.)

Missing argument

Directive argument is expected.

Bad argument

Invalid argument

Legal directive argument is expected.

Directive Error Messages

Illegal character in directive arguments

Directives must be carefully described. (See the Assembler chapter.)

Equate directive requires a label

A label must precede all equate directives.

Example:

```
$4000 equ tempvar ; ERROR
tempvar equ $4000 ; OK
```

**Invalid expression for equate directive
Need absolute value here**

An absolute value is a constant expression.

An absolute value must follow a set directive.

Examples:

```
rex equ 1 ;           OK
spot equ 2 ;          OK
fido equ rex+spot ;   OK
rover equ rex+(a4)     ERROR
```

Set directive requires a label

(see: Equate directive requires a label)

Invalid expression for set directive

(see: Invalid expression for equate directive)

**Name already defined
Multiply defined symbol**

Every label must have a unique name.

**Register list directive requires a label
Need register list here
REG directive must contain register list**

Register list directives must be specified as follows:

label *reg* *register_list*

Macros must be defined as:

[*label*] **macro** *symbol*

The macro name is the label preceding the MACRO directive or the symbol following it. Every macro name must be unique.

Macro end out of place

The ENDM directive must be used in conjunction with the MACRO directive. All directives and opcodes must be preceded by at least one *space* or *tab*.

Need symbol for definition check

Requires two strings

Right side of test must match left side

Illegal character in conditional

Premature end of line in conditional string

These errors pertain to conditional directives.

Conditional end directive out of place

endc must be used in conjunction with an if directive.

Need file name here

Unable to open include file

Include directives must be specified as:

`include <filename>`

`include "filename"`

Need 'code' or 'data' here

The near and far directives must specify 'code' or 'data'

Examples:

`near code`

`far data`

Need second argument as well

The cnop directive requires two arguments.

Invalid expression for defined storage

The value of the size field in the **ds** directive must be an absolute expression.

Unimplemented assembler directive

The assembler does not understand. Check your syntax.

Syntax Errors

```
bad '('  
Missing ')'; Where's the ')'; Missing trailing paren-  
thesis;  
Missing ']'; illegal ']';  
illegal '['  
bad syntax  
extra characters on line!  
Unknown token in expression  
Unknown opcode or directive  
Illegal character in string  
Bad character in line
```

Please check your typing.

Linker Error Messages

This section discusses the error messages that the linker may display as it creates an executable program. Below is a summary of the error messages, followed by a more detailed explanation of each.

Note: Only one file type option is permitted.

Explanation of Linker Error Messages

When started, the linker first displays a message on the screen that indicates that the linker is loaded and running. If everything goes well, the linker prints several messages on the screen listing the sizes of the program segments; then the linker finishes. The linker may encounter an error while it is running, in which case it sends a message to the screen.

Errors are reported at a variety of points during the linking process. It generates an executable program in two stages, known as pass 1 and pass 2. The size messages are printed at the end of pass 1, so any errors occurring after that are detected during pass 2 of the linker.

Following is a list of the messages that the linker generates in response to an error. The messages are grouped according to the source of the errors that cause them. Elements that are variable are enclosed by angled brackets.

Command Line Errors

Unknown option 'c'

You have used an option letter(c) that the linker does not recognize. The linker ignores only the letter; it preserves everything else on the command line and the linker tries to execute what it can interpret. See the **Linker** chapter for a list of options that are supported.

Too few arguments in command line

Several of the linker options have an associated value or name, such as -b 2000. If a needed value is missing, the linker displays this message and dies.

No input given!

The linker quits immediately if it is not given any input to process.

Cannot have nested -f options.

At the command line level any number of command files can be specified by using the -f option. However, none of these files can contain the -f option. A command file cannot invoke another command file

Too few arguments in -f file: *filename*

An option letter specified in the file, *filename*, requires a value or name to follow it. If an option appears at the end of the file, its associated value may not appear back on the command line.

Multiple entry points defined.

Multiple global symbols have been found that have the same name.

Invalid overlay number

Overlays must be positive integers.

I/O Errors**Cannot open *filename*, err = *errno***

If any file in the command line cannot be opened, this message is sent to the screen, specifying the *filename* and the current value of *errno*.

Cannot open -f file: *filename*

A file given with option -f cannot be opened.

I/O error (*errno*) reading/writing output file

An error reading or writing the output file probably means there is no more disk space available. In particular, a block of the output file was written to

disk and then could not be read back. The current value of *errno* is given in these messages.

Cannot write output file

Cannot create output file: *filename*

This message usually indicates that all available directory space on the disk has been exhausted.

Cannot create symbol table output

Error creating .map file

Cannot create debugger symbol file *filename*

Error creating symbol listing file

Option *-t* is given in the command line, but the file containing the linkage symbol table cannot be written to disk. It is possible that there is no more space on the disk.

Corrupted object files

Object file is bad!

This is the most explicit indication that an object file in the linkage is corrupted. Recompile and assemble the source file. A bad object file will not be discovered until the second pass of the linker.

Invalid operator in evaluate *hex_value*

Unless you changed the object code by hand, the file is corrupted.

Library format is invalid!

A library in the linkage has been corrupted.

Cannot read module from input on pass2
...or cannot find *symbol*, *symbol name*, on pass two

Either message indicates that a module is corrupted between pass 1 and pass 2. On a multiuser system, it is possible that another user changed the file while the linker was running. Otherwise, the error is probably due to a hardware failure.

Not an object file

A file given to the linker does not contain relocatable object code that `ln` can process. For instance, a source file may have been included in the link.

Bad symbol typing information
Line displacement too large
Symbol name too long

Source-level debugging information is in error. Make sure you have a current version of the linker and the compiler.

Errors in Use of Memory

Insufficient memory!

The linkage process needs memory space for `ln` and global and local symbol tables, and approximately 5K for buffers. Just as with compilation, most memory use is devoted to the program software and symbol tables. Since `ln` is not especially large, only an extremely complicated linkage might run out of memory.

Too many symbols!

This is another way of saying that not enough memory is available for the symbol tables needed for the linkage.

Errors Arising From Source Code

Undefined symbol: *symbol_name*

A global symbol name remained undefined. This is commonly a function that has been referenced in the source code but not included anywhere in the link.

symbol_name multiply defined

A global symbol is defined more than once. For instance, if two functions are accidentally given the same name, this message is generated.

pass1(hex_value) and pass2(hex_value) values differ: *or symbol type differs on pass two: symbol_name*

Either of these errors may be generated during pass 2 when error 24 appeared in pass 1. They may be considered a confirmation of what was discovered in pass 1 of the linker.

Branch out of range @pc=<addr>

A **branch** or **jump** instruction has a target address that is beyond its range. This error should not be generated from C programs.

Short branch to next location @pc=addr

The 68k processor does not accept instructions of this type. This error should not occur, since the Manx assembler detects such instructions and removes them from the object code.

Entry point must be in root segment

The entry point for a program must be in the program root segment. For example, this error would be generated if you tried to link the "hello, world" program with the command:

```
ln hello.o +0 -lc
```

Entry point must be in first 64K of program

Not only must a program's entry point be in the root segment, it must also be in the first 64k bytes of this segment. The reason for this is that a 16-bit field in the jump table points to the entry point.

Attempt to store out of bounds

This error should not occur. It indicates a linker bug.

Program is too large to link

This error should not occur. It indicates a linker bug.

Attempt to perform relocation in overlay code

The only segments of a program that can contain addresses that must be relocated when the program is loaded are the program root code segment and its initialized data segment. The relocation of these two segments is performed when the program is first loaded, by the Manx-supplied startup code.

data ref to overlay code not in jump table

This error is caused by C programs that attempt to initialize a global pointer to a static function, where the function is contained in an overlay. Such initialization is permitted when the function is located in the root segment, but not when it is in another segment.

BigCrt0.o must be in root segment

Attempt to link large data or large segments without appropriate starting code. Place **BigCrt0.o** early in your **ln** command.

Absolute reference from segment to segment

Segments may only reference other segments via the jump table. Check your assembler code, the compiler will never generate these.

Excess number of segments

Linker internal overflow. Try using fewer and larger segments.

Can't do relocation from data to nonroot code

The only permissible data references are via the jump table. Therefore data can only be assigned the address of a function's entry point.

data reference out of range—remake with large data model

Initialized data plus uninitialized data exceeds 32K. Recompile all C source modules with **-md**, reassemble assembly modules with **-d**, and relink with **cld.lib** (large data). An alternative approach would be to dynamically allocate some of your data at run time.

- #asm/ #endasm
 - See embedded assembly
- #include files
 - pre-compiled, 2-6 - 2-7
 - search order, 2-6
 - searching for, 2-5
- #pragmas
 - See pragmas
- * operand as argument to print, db, 7-35
- .dbg file, 5-16
- .dbinit file, db, 7-7
- == command, db, 7-43
- ? command, db, 7-44
- @function, db, 7-12
- __FILE__, 3-13
- __FUNC__, 3-13
- __LINE__, 3-13
- __INT32, 3-14
- __LARGE_CODE, 2-26, 3-14
- __LARGE_DATA, 2-26, 3-14

A

- abnormal termination of program, db, 7-4
- accessing variables
 - from C and assembly, 8-14
- add command, db, 7-14
- ADDR
 - definition, 7-11
 - examples, 7-13
- addressing, db commands, 7-15
- adi command, db, 7-14
- adl command, db, 7-14
- adp command, db, 7-14
- adr command, db, 7-14
- adump utility, 6-3

- aggregate types
 - arrays, 3-10
 - structures, 3-10
- ai command, db, 7-15, 7-32
- ak command, db, 7-15
- al command, db, 7-16
- am command, db, 7-17
- amicall
 - See pragmas
- Amiga ROM Kernal Manual, 8-18
- Amiga specific commands, db, 7-14
- an command, db, 7-17
- ANSI language specifications, 3-1
 - examples, 3-8
 - auto aggregate initialization, 3-10
 - extended syntax, 3-7
 - features supported, 3-6
 - function prototypes, 3-6
 - K&R versus ANSI, 3-7
 - overview, 1-1, 2-36
 - predefined symbols, 3-13
 - See also prototypes
 - size_t, 3-13
 - sizeof and size_t, 3-13
 - trigraphs, 3-11
 - wide strings and literals, 3-10
- ANSI preprocessor features
 - line continuation, 3-15
 - predefined symbols, 3-13 - 3-14
 - string concatenation, 3-15
- ap command, db, 7-17
- aq command, db, 7-18
- ar command, db, 7-19
- arcv utility, 6-2
- array, initialization, 3-10
- as command, db, 7-19
- assembler
 - definition, 4-1
 - syntax, 4-2
 - accessing modules, 8-12

- arguments to subroutines, 4-21
- embedded with C, 2-44, 4-21 - 4-22
- See also embedded assembly
- executable instructions, 4-8
- execution environment, 4-2
- file creation, 2-4
- functions callable from C, 8-12 - 8-16
- INCLUDE environment variable, 4-4
- include files, 4-4 - 4-5
- input file, 4-2
- listing file, 4-3
- memory models, 4-5
- object code file, 4-3
- operating instructions, 4-2
- optimizations, 4-3 - 4-4
- output control, 2-15
- processor support, 4-2
- register conventions, 4-21
- searching for include files, 4-4
- source program structure, 4-8 - 4-14, 4-16, 4-18 - 4-20
- source statements, 4-1
- using with C source code, 2-44 - 2-45, 8-12, 8-15
- assembler directives, 4-10, 4-12 - 4-14, 4-16, 4-18 - 4-20
 - blanks, 4-10
 - bss, 4-16
 - clist, 4-10
 - cnop, 4-11
 - cseg, 4-11
 - dc - define constant, 4-12
 - dcb - define constant block, 4-12
 - ds - define storage, 4-13
 - dseg, 4-11
 - else, 4-15
 - end, 4-13
 - endc, 4-15
 - endm, 4-17 - 4-18
 - entry, 4-13
 - equ, 4-13
 - equr, 4-14
 - even, 4-14

- fail, 4-14
- far code, 4-14
- far data, 4-14
- freg, 4-14
- global, 4-16
- if, 4-15
- ifc, 4-15
- ifd, 4-15
- ifnc, 4-15
- ifnd, 4-15
- include, 4-16
- list, 4-17
- machine, 4-17
- macro, 4-17 - 4-18
- mc68851, 4-18
- mc68881, 4-18
- mexit, 4-18
- mlist, 4-17
- near code, 4-16
- near data, 4-16
- noclist, 4-10
- nolist, 4-17
- nomlist, 4-17
- other ifs, 4-16
- public, 4-19
- reg, 4-19
- section, 4-19
- set, 4-19 - 4-20
- ttl, 4-20
- xdef, 4-20
- xref, 4-20
- assembler error messages
 - opcode summary list, 9-7 - 9-9
 - explanations, directive errors, 9-66 - 9-69
 - explanations, opcode errors, 9-62 - 9-66
 - explanations, syntax errors, 9-69
- assembler options
 - summary list, 4-6
 - c, 4-5, 4-7
 - d, 4-5, 4-7

- i, 4-4 - 4-5, 4-7
- l, 4-3, 4-7
- n, 4-3
- o, 4-7
 - o example, 4-3
- at command, db, 7-19
- auto aggregate initialization, 3-10
- available memory
 - segmentation, 8-8

B

- backtracing, db, 7-6
- bb command, db, 7-20
- bc command, db, 7-21
- bd command, db, 7-21
- bh command, db, 7-22
- bitfields, 3-5
- bl command, db, 7-20
- bq command, db, 7-22
- br command, db, 7-23
- branches
 - optimizations, 4-3
- breakpoints, db, 7-4
 - commands, 7-5
 - skip count, 7-4
- bs command, db, 7-23
- bt command, db, 7-24
- bu command, db, 7-24
- bw command, db, 7-20

C

- C and assembly
 - accessing variables, 8-14
 - stack operations, 8-13
- c.lib, 8-11
- CCOPTS environment variable
 - See environment variables
- char type, 3-4
- characters

- long constant, 3-12
 - non-alphanumeric, 3-11
 - wide, 3-10
- chip or processor control, 2-15
- clear commands, db, 7-25
- cmdlist, definition, 7-14
- cmp utility, 6-4
- cnm utility, 6-5 - 6-9
 - options, 6-6
 - symbol format, 6-7
 - symbol types, 6-7
 - type codes, 6-7 - 6-9
- code segmentation, 2-14
- code segments, 8-2
 - location in memory, 8-3
 - size, 8-4
- command breakpoints, db, 7-5
- command line errors
 - See linker error messages
- command summary, db
 - See db commands
- compatibility, 3-1
 - Amiga Workbench, 3-1
 - UNIX, 3-1
 - with other Aztec compilers, 3-1
 - XENIX, 3-1
- compiler
 - See also libraries
 - _LARGE_CODE, defining, 2-26
 - _LARGE_DATA, defining, 2-26
 - aborting, 2-2
 - code segment limits, 2-14
 - converting data, 2-43
 - declared functions, 2-41 - 2-42
 - displaying error messages, 2-34
 - embedded assembly, 2-44
 - example, 2-44
 - enums, 2-27
 - error handling, 2-34
 - fatal errors, 2-34

- for loops, 2-29
- handling errors, 2-34
- input files, 2-2
- internal storage of numeric data, 2-43
- invoking, 2-1
- line continuation, 2-44
- memory models, 2-8 - 2-14
- mixing memory models, 2-13
- See also memory models
- nonfatal errors, 2-34
- output files, assembly language, 2-4
- output files, object code, 2-3
- pointer considerations, 2-41
- pointer use, rules, 2-41
- See also pointers
- pre-compiled #include files, 2-6 - 2-7
- predefined symbols, 3-14
- prepended underscore use in db, 7-3
- prototypes, 2-28
- searching for #include files, 2-5
- selecting a memory model, 2-12
- syntax, 2-1
- table manipulation options, description, 2-34
- trigraph option, 3-12
- trigraphs, 2-28
- utility options - description, 2-23
- compiler error messages
 - summary list, 9-2 - 9-6
 - fatal error summary list, 9-7
 - explanations, 9-12 - 9-57
 - explanations, fatal errors, 9-58 - 9-62
 - explanations, internal errors, 9-62
- compiler options, 2-14
 - summary list, 2-17 - 2-20
 - 3 option, 2-17, 2-21 - 2-22
 - 5 option, 2-17, 2-21 - 2-22
 - a option, 2-2, 2-4 - 2-5, 2-17
 - a examples, 2-4
 - at option, 2-4 - 2-5, 2-17
 - bd option, 2-17

- bs option, 2-17, 5-16
- c2 option, 2-17, 2-22
- d option, 2-17, 2-22 - 2-23
- e option, 2-21
- f8 option, 2-17, 2-24
- fa option, 2-17, 2-23
- ff option, 2-17, 2-23
- fm option, 2-17, 2-24
- hi option, 2-7, 2-17, 2-24
 - hi examples, 2-7
- ho option, 2-7, 2-17, 2-24
 - ho examples, 2-7
- i option, 2-5, 2-17, 2-25
- k option, 2-17, 2-21, 2-25
- ma option, 2-17
- mb option, 2-18, 2-26
- mc option, 2-12 - 2-13, 2-18, 2-26
- md option, 2-12 - 2-13, 2-18, 2-26
- me option, 2-18
- mm option, 2-18, 2-26
- ms option, 2-18
- o option, 2-18, 2-27
- pa option, 2-18, 2-27
 - pa examples, 2-15
- pb option, 2-18, 2-27
- pc option, 2-18
- pe option, 2-18, 2-27
- pl option, 2-18, 2-27
- po option, 2-18
- pp option, 2-18
- ps option, 2-18, 2-28, 2-41
 - ps examples, 2-15
- pt option, 2-18, 2-28
- pu option, 2-18 - 2-19
- qa option, 2-19, 2-28
- qf option, 2-1, 2-19, 2-29
- qp option, 2-19, 2-29
- qq option, 2-19, 2-29
- qs option, 2-19, 2-29
- qv option, 2-19, 2-29

- r4 option, 2-19
- sa option, 2-19
- sb option, 2-19
- sf option, 2-19, 2-29
- sm option, 2-19
- sn option, 2-19, 2-30
- so option, 2-19
- sp option, 2-19, 2-30
- sr option, 2-19, 2-30
- ss option, 2-20, 2-30
- su option, 2-20, 2-31
- wa option, 2-20, 2-31
- wd option, 2-20, 2-31
- we option, 2-20
- wl option, 2-20, 2-32
- wn option, 2-20, 2-32
- wo option, 2-20, 2-32
- wp option, 2-20, 2-32
- wq option, 2-20, 2-33
- wr option, 2-20, 2-33
- ws option, 2-20
- wu option, 2-20, 2-33
- ww option, 2-20, 2-34
- z option, 2-21
- compiler options, turning off, 2-15
- const, 3-5
- constant
 - definition, 7-10
- continuation, line, 3-15
- cs command, db, 7-25
- current directory
 - loading db files from, 7-4

D

- d command, db, 7-26
- data segments
 - location in memory, 8-4
 - size limitations, 8-4
- data types, 3-2
 - bitfields, 3-5

- example, 3-5
- characters, 3-4
 - example, 3-4
- const, 3-5
- enums, 3-5
- integers and long integers, 3-3
 - example, 3-3
- structures, 3-4
 - example, 3-4
- volatile, 3-5
- data, converting, 2-43
- db
 - See also db terms
 - .dbinit file, 7-7
 - @function, 7-12
 - abnormal termination of program, 7-4
 - addressing commands, 7-15
 - Amiga specific commands, 7-14
 - backtracing, 7-6
 - basic commands, 7-2
 - breakpoint skip count, 7-4
 - breakpoints, 7-4 - 7-5
 - code symbols, 7-3
 - command format, 7-2
 - command vs table breakpoints, 7-5
 - compiler prepended underscore, 7-3
 - creating macros, 7-6
 - data symbols, 7-3
 - default radix setting, 7-10
 - defining breakpoints, 7-4
 - desc_code list, print option, 7-37
 - display code and data symbols, 7-3
 - entering a function, 7-6
 - environment variables, 7-7
 - examining memory, 7-2
 - examining variables, 7-4
 - exiting, 7-39
 - exiting a function, 7-6
 - loading programs, 7-3
 - loading programs not in current directory, 7-4

- loading symbols, 7-4
- macros, 7-6
- memory change breakpoints, 7-5
- modifying memory, 7-2
- overlay breakpoints, 7-8
- print command * operand, 7-35
- print command format string, 7-33
- print count specifier, 7-33
- print desc_code, 7-33
- program control, 7-4
- program termination, 7-4, 7-8
- referencing symbols by location, 7-3
- referencing symbols by name, 7-3
- removing memory breakpoints, 7-6
- requirements, 7-1
- scatter loading support, 7-8
- segment loading and unloading, 7-9
- setting memory breakpoints, 7-6
- single-stepping, 7-2
- skip counter, 7-5
- special characters, 7-11
- starting db, 7-3, 7-7
- startup events, 7-7
- stopping display, 7-9
- stopping the program, 7-9
- symbol table generated by linker, 7-3
- symbol table hunks, 7-3
- trace mode, 7-6
- unassembly, 7-2
- viewing code and data symbols, 7-3
- db command, db, 7-26
- db commands
 - summary list, 7-45
 - Special character commands, 7-43
 - < - redirect input, 7-43
 - == - display expression, 7-43
 - > - log output to file, 7-43
 - >> - log commands only, 7-43
 - ? - help, 7-2, 7-7, 7-12, 7-44
 - Amiga specific commands, 7-15

add - display device list, 7-14
 adi - display interrupt list, 7-14
 adl - display library list, 7-14
 adp - display port list, 7-14
 adr - display resource list, 7-14
 ai - display task info, 7-15, 7-32
 ak - kill current task, 7-15
 al/aL - debug next task w/o symbols, 7-16
 am - amt of free memory in system, 7-17
 an/aN - new window and symbols, 7-17
 ap/aP - debug a crashed program, 7-17
 aq - close all windows, 7-18
 ar - allow current task to resume, 7-19
 as/aS - select a task to debug, 7-19
 at - display all tasks, 7-19
 Breakpoint commands, 7-2
 bb - set byte breakpoint, 7-20
 bc/bC - clear breakpoint, 7-21
 bd - display breakpoints, 7-21
 bh - set memory hash breakpoint, 7-22
 bl - set long breakpoint, 7-20
 bq - toggle low memory checksum, 7-22
 br - reset breakpoint counter, 7-23
 bs - set or modify a breakpoint, 7-23
 bt/bT - toggle trace mode flag, 7-24
 bu - user defined breakpoint, 7-24
 bw - set word breakpoint, 7-20
 Clear symbol table commands, 7-25
 cs - clear symbol table, 7-3, 7-25
 Display commands, 7-26
 d - display memory in last format, 7-26
 db - display bytes in memory, 7-26
 dc - display all code symbols, 7-26
 dd - display data symbols, 7-27
 dg - display global values, 7-27
 dl - display memory in longs, 7-26
 ds - display stack backtrace, 7-27
 dw - display memory in words, 7-26
 Memory commands, 7-29
 ma - allocate some memory, 7-29

- mb - modify bytes of memory, 7-30
- mc - compare memory, 7-30
- mf - fill memory, 7-31
- ml - modify double words of memory, 7-30
- mm - move memory, 7-31
- ms - search memory, 7-31
- mw - modify words of memory, 7-30
- Miscellaneous commands, 7-28 - 7-29
- n - change radix, 7-32
- g/G - execute the program, 7-28, 7-32, 7-34 - 7-42
- db terms
 - addr, 7-13
 - addr and *addr, 7-11
 - cmdlist, 7-14
 - constant, 7-10
 - expr, 7-9
 - period, 7-12
 - range, 7-13
 - registers, 7-10
 - term, 7-9
- dc command, db, 7-26
- dd command, db, 7-27
- debugging control, 2-15
- declared functions, 2-41 - 2-42
- default radix setting, db, 7-10
- default settings, 2-14
- defining breakpoints, db, 7-4
- desc_code list, db, 7-37
- device drivers, 5-1
- dg command, db, 7-27
- directive error messages
 - See assembler error messages
- directives, assembler
 - See assembler directives
- display task information, db, 7-15
- dl command, db, 7-26
- ds command, db, 7-27
- du utility, 6-10
- dw command, db, 7-26

E

- embedded assembler
 - examples, 2-44, 8-12
 - source, 8-15
- enums, 2-27, 3-5
- environment variables
 - See also specific variable name
 - CCOPTS, 2-14 - 2-16, 2-21 - 2-22
 - CCTEMP, 2-4
 - CLIB, 5-4, 5-11
 - db, 7-7
 - displaying, 6-28
 - INCLUDE, 2-5 - 2-6, 4-4 - 4-5
 - setting, 6-28
- error handling, compiler, 2-34
- error messages, assembler
 - See assembler error messages
- error messages, compiler
 - See compiler error messages
- error messages, linker
 - See linker error messages
- escape sequences, 3-12
- examining variables, db, 7-4
- executable file, 5-4, 5-8
- executable instructions
 - assembler, 4-8 - 4-10
 - labels, 4-8
 - operands, 4-9 - 4-10
 - operations, 4-8 - 4-9
- executable programs, 5-1, 5-7
- execution environment
 - assembler, 4-2
- exiting a function, db, 7-6
- exiting db, 7-39
- EXPR, definition, 7-9
- extern keyword
 - use with global variables, 8-15
- external functions, 8-12

F

- floating point control, 2-15
- floating point support, 8-18
 - compile requirements, 8-19
 - compiler options, 2-23
 - descriptions of, 8-19
 - formats, 8-19
 - link requirements, 8-19
 - register usage, 8-13
- function
 - K&R versus ANSI, 3-7
 - structure arguments, 3-11
- function calls, 8-13
- function calls and returns, 8-13
- function definition, 3-7
- function entering, db, 7-6
- function prototypes, 3-9
 - See prototypes
- functions.h, 2-37 - 2-38

G

- g command, db, 7-28
- geta4 function, 8-18
- global data, 8-9
- global variables, 8-14

H

- hd utility, 6-11
- heap segments, 8-3
 - size of, 8-4
- help command, db, 7-7
- hunk data, 5-14, 8-6 - 8-8

I

- I/O errors
 - See linker error messages
- INCLUDE environment variable
 - See environment variables
- include files

- i option, 4-4
- environment variable, 4-5
- search order, 4-5
- searching for, 4-4
- initialized data, 8-2, 8-4
- input files, 2-2, 5-8
 - assembler, 4-2
 - source filename extensions, 2-2 - 2-3
- interrupt handlers
 - examples, 8-17
 - compile options, 8-17
 - defined, 8-16
 - int_end, 8-17
 - int_start, 8-17
 - preserving registers, 8-16 - 8-17
- interrupt service routines, 8-16

J

- jsr optimizations, 4-4
- jump table, segmentation, 8-6

K

- keywords
 - list, 3-2
 - extern, 8-15

L

- Language specs
 - See also data types
 - See also ANSI preprocessor features
 - ANSI syntax, 3-7
 - data type list, 3-3
 - escape sequences, 3-12
 - keywords list, 3-2
 - long character constants, 3-12
 - operator list, 3-2
 - preprocessor directive list, 3-2
 - register variables, 3-13
 - structure assignment, 3-11

- structure passing, 3-11
 - symbol names, 3-13
- LARGE CODE, 3-14
- LARGE DATA, 3-14
- lb utility, 5-3, 6-12 - 6-22
 - adding modules, 6-18 - 6-19
 - advanced features, 6-17
 - basic features, 6-14
 - creating a library, 6-14
 - default extension, defining, 6-22
 - deleting modules, 6-19 - 6-20
 - extracting modules, 6-21
 - function code options, 6-12
 - getting arguments from a file, 6-16
 - library argument, 6-12
 - mod arguments, 6-13
 - moving modules, 6-19
 - naming modules, 6-15
 - options argument, 6-12
 - ord, 6-27
 - qualifier options, 6-13
 - reading arguments from a file, 6-13
 - rebuilding a library, 6-22
 - replacing modules, 6-20
- libraries, 2-12, 2-35, 5-3, 5-9 - 5-10, 8-10
 - examples, 5-5 - 5-6, 8-11
 - adding modules, 6-18 - 6-19
 - c.lib, 5-9, 8-10
 - creating, 6-14
 - deleting modules, 6-19 - 6-20
 - extracting modules, 6-21
 - lb utility, 6-17 - 6-22
 - m.lib, 5-9
 - moving modules, 6-19
 - naming conventions, 8-11
 - order of modules, 5-5 - 5-7, 6-15
 - rebuilding, 6-22
 - replacing modules, 6-20
- line continuation, 2-44
- linker

- examples, 5-3 - 5-4, 5-8, 5-10
- See also environment variables
- CLI programs, 5-1
- CLIB environment variable, 5-4, 5-11
- device drivers, 5-1
- executable file, 5-8
- executable programs, 5-1
- generates db symbol table, 7-3
- hunk data, 5-14
- input files, 5-8
- introduction, 5-2
- libraries, 5-3, 5-5, 5-9 - 5-10
- relocatable object format, 5-1
- starting, 5-7
- syntax, 5-7
- the process, 5-2 - 5-4
- use, 5-7 - 5-8, 5-10
- workbench programs, 5-1
- linker error messages, 9-10
 - command line summary list, 9-10 - 9-11
 - explanations, command line errors, 9-70 - 9-71
 - explanations, I/O errors, 9-71 - 9-73
 - explanations, memory errors, 9-73
 - explanations, source code errors, 9-74 - 9-76
- linker options, 8-3
 - +a long word boundaries, 5-10
 - +c chip memory, 5-15, 8-3
 - +f fast memory load, 5-10, 5-15, 8-3
 - +l, 5-10
 - +o, 5-11
 - +s val, 5-11
 - a, 4-11
 - f, 5-10, 5-12, 8-10
 - g source level debug, 5-10, 5-16
 - l name, 5-10 - 5-11, 8-10
 - m, 5-10
 - o, 5-8, 5-11, 8-10
 - q, 5-11
 - t, 5-11
 - v, 5-11

- w, 5-11
- listing file, assembler, 4-3
- literals, wide, 3-10
- loading programs, db, 7-3
- loading symbols, db, 7-4
- long character constant, 3-12
- ls command, db, 7-29
- ls utility, 6-23

M

- m.lib, 8-11
- m8.lib, 8-11
- ma command, db, 7-29
- ma.lib, 8-11
- macro calls
 - definition, 4-20
- macros
 - creating in db, 7-6
- mb command, db, 7-30
- mc command, db, 7-30
- memory change breakpoints, db, 7-5
- memory errors
 - See linker error messages
- memory models, 2-8 - 2-14
 - assembler, large code, 4-5, 4-7
 - assembler, large data, 4-5, 4-7
 - code segmentation, 2-14, 8-4
 - control of, 2-15
 - large code vs small code, 2-9 - 2-11
 - large data vs small data, 2-8 - 2-9
 - libraries, 2-12
 - multimodule programs, 2-13
 - selection for use by a module, 2-12
 - using the correct libraries, 8-11
- mf command, db, 7-31
- mf.lib, 8-11
- mkarvc utility, 6-25
- ml command, db, 7-30
- mm command, db, 7-31
- movem

- optimizations, 4-4
 - reg directive, 4-19
- ms command, db, 7-31
- multi-tasking, 8-17
 - examples, 8-18
 - use of a4 register, 8-18
- multimodule programs, 2-13
- mw command, db, 7-30

N

- n command, db, 7-32
- naming conventions
 - C and assembly, 8-12
- numbering segments, 8-9
- numeric data
 - internal storage, 2-43

O

- obd utility, 6-26
- object code file
 - assembler, 4-3
 - creation, 2-3 - 2-4
- object modules, 8-11
- opcode error messages
 - See assembler error messages
- operating instructions
 - assembler, 4-2
- operator list, 3-2
- optimizations
 - assembler, 4-3 - 4-4
 - branches, 4-3
 - control, 2-15
 - jsr, 4-4
 - movem, 4-4
 - volatile, 3-5
- ord utility, 5-6, 6-27
- output control, 2-15
- output files, 2-3 - 2-4
- overlay breakpoints, db, 7-8

P

- p command, db
 - See print command, db
- parser control, 2-15
- pointer considerations, 2-41
- pointers
 - declare function arguments that are pointers, 2-42
 - declare functions that return pointers, 2-41
 - variables and constants as arguments, 2-42
- portability, 3-1
 - integers, 3-3
- pragmas
 - definition, 2-36
 - a6 register, 2-36
 - amicall, 2-36 - 2-40
 - Amiga exec calls, 2-35
 - calling resident libraries, 2-36 - 2-38
 - creating resident libraries, 2-38 - 2-39
 - libcall, 2-40
 - regcall, 2-40
 - register function calls, 2-39
 - syscall, 2-40
- pre-defined symbols, 3-13
- precompiled #include files, 2-6
- predefined symbols
 - __FILE__, 3-14
 - __FUNC__, 3-14
 - __LINE__, 3-14
 - _INT32, 3-14
 - _LARGE_CODE, 3-14
 - _LARGE_DATA, 3-14
 - AZTEC_C, 3-14
 - MCH_AMIGA, 3-14
 - MPU68000, 3-14
 - VERSION, 3-14
- preprocessor
 - concatenation, 3-15
 - directives, 3-2
 - features, 3-13
- preserving registers, 8-15

- See registers
- print command, db, 7-2
 - examples, 7-35 - 7-38
 - count specifier, 7-33
 - desc_codes, 7-33, 7-38
 - format items, 7-33
 - format string, 7-33
 - indir, 7-34
 - rpt parameter, 7-35
- processor support
 - assembler, 4-2
- program control, db, 7-4
- program organization, 8-2
- program termination, db, 7-4, 7-8
- prototypes, 2-28
 - function call, 3-7
 - function declarations, 3-6, 3-9
 - function definition, 3-7

Q

- q command, db, 7-39
- QuikFix, 2-1 - 2-2, 2-19, 2-29, 2-34

R

- r command, db, 7-40
- RAM, 2-36
- range
 - definition, 7-13
- register conventions
 - assembler, 4-21
- registers
 - a4, 2-13, 2-26, 8-17
 - a5, 2-30
 - a6, 2-36
 - control of, 2-15
 - definition, 7-10
 - floating point, 4-14
 - preserving, 8-15 - 8-17
 - reg directive, 4-19

- restoring, 4-22
- stack pointer, 4-21
- usage, 8-15
- variables, 3-13
- relocatable object format, 5-1
- requirements, db, 7-1
- reselecting segments, 8-10
- resident library
 - example, 2-39
 - Amiga functions, 2-37
 - calling, 2-36 - 2-37
 - creating, 2-38
- ROM, 2-36
- ROM applications
 - suppressing dseg generation, 2-26
- root segment
 - startup code, 8-7

S

- s command, db, 7-40
- s.lib, 8-11
- scatter loading support, db, 7-8
- segment loading and unloading, db, 7-9
- segmentation, 8-5
 - examples, 8-9 - 8-10
 - creating segments, 8-8
 - function exclusivity, 8-5
 - global data, 8-9
 - hunk data, 8-6, 8-8
 - jump table, 8-6
 - linker requirements, 8-9
 - linking startup code, 8-7
 - loading segments, 8-5
 - maximum number of segments, 8-9
 - memory usage, 8-8
 - numbering segments, 8-9
 - program design, 8-5
 - reselecting segments, 8-10
 - segload, 8-6
 - segment construction, 8-7

- segment data table, 8-8
- segment size, 8-4
- segment size limits, 8-4
- segmented code, 8-5, 8-8
- segments, 8-2 - 8-3
- static data, 8-9
- unloading segments, 8-6
- set utility, 6-28
- setdate utility, 6-29
- setting and removing memory change breakpoints, db, 7-6
- size types, 3-3
- size_t, 3-13
- sizeof, 3-4, 3-13
- skip counter, db, 7-5
- source code errors
 - See linker error messages
- source filename extensions, 2-2 - 2-3
- source program structure
 - comments, 4-8
 - executable instructions, 4-8
- special characters, db, 7-11
- stack segments, 8-3
 - C and assembly, 8-13
 - CLI default value, 8-4
 - function calls, 8-13
 - stack size, 8-4
- starting, db, 7-7
- startup code, 8-7
 - linking, 8-7
 - location, 8-7
 - root segment requirements, 8-7
- static data, 8-9
- stopping display, db, 7-9
- stopping program execution, db, 7-9
- string concatenation, 3-15
- strings
 - long, 3-15
 - wide, 3-10
- structure
 - alignment, 3-4

- initialization, 3-10
- subroutines, arguments to
 - floating point, 4-21
 - returning from a C function, 4-22
 - variable names, 4-21
- subtasks, creating, 8-17
 - example, 8-18
- symbol names, 3-13
- symbol table hunks, db, 7-3
- syntax error messages
 - See assembler error messages

T

- table breakpoints, db, 7-5
- tables, 2-34
- TERM, definition, 7-10
- termination of program, db, 7-4
- toggle, 2-14
- trace mode, db, 7-6
- trigraphs, 2-28, 3-11
- turning off compiler options, 2-15

U

- u command, db, 7-41
- underscore
 - referencing symbols in db, 7-3
- uninitialized data, 8-3 - 8-4
- UNIX compatibility, 3-1
- unsigned characters, 3-4
- utilities:
 - adump - Amiga object module dump, 6-3
 - arcv - text file archiver, 6-2
 - cmp - file comparison, 6-4
 - cnm - display file info, 6-5 - 6-9
 - du - disk usage, 6-10
 - hd - hex dump, 6-11
 - lb - object file librarian, 6-12 - 6-22
 - ls - list files and directories, 6-23
 - mkarcv - text file archiver, 6-25

obd - list object code, 6-26
ord - sort object module, 6-27
set - set environment variable, 6-28
setdate - set date and time, 6-29

V

v command, db, 7-42
variables, 8-3
volatile, 3-5

W

warning control, 2-15
wide literals, 3-10
wide strings, 3-10
workbench, 8-19
 examples, 8-21
 creating programs, 8-19 - 8-22
 starting up, 8-21

X

x command, db, 7-42
XENIX compatibility, 3-1